

Коичи Мацуда
Роджер Ли

WebGL:

**программирование
трехмерной графики**



Коичи Мацуда, Роджер Ли

WebGL: **программирование** **трехмерной графики**

WebGL Programming Guide:

Interactive 3D Graphics
Programming with WebGL

Kouichi Matsuda

Rodger Lea

◆◆ **Addison-Wesley**

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

WebGL: программирование трехмерной графики

Коиичи Мацуда (Kouichi Matsuda)
Роджер Ли (Rodger Lea)



Москва, 2015

УДК 004.738.5:004.4WebGL
ББК 32.973.202-018.2
МЗЗ

МЗЗ Коичи Мацуда, Роджер Ли

WebGL: программирование трехмерной графики. / Пер. с англ. Киселев А. Н. – М.: ДМК Пресс, 2015. – 494 с.: ил.

ISBN 978-5-97060-146-4

WebGL является новой веб-технологией, позволяющей использовать в браузере преимущества аппаратного ускорения трехмерной графики без установки дополнительного программного обеспечения. WebGL основана на спецификации OpenGL и привносит новые концепции программирования трехмерной графики в веб-разработку.

Снабженная большим количеством примеров, книга показывает, что овладеть технологией WebGL совсем несложно, несмотря на то, что она выглядит незнакомой и инородной. Каждая глава описывает один из важнейших аспектов программирования трехмерной графики и представляет разные варианты их реализации. Отдельные разделы, описывающие эксперименты с примерами программ, позволяют читателю исследовать изучаемые концепции на практике.

Издание предназначено для программистов, желающих научиться использовать в своих веб-проектах 3D-графику.

УДК 004.738.5:004.4WebGL
ББК 32.973.202-018.2

Original English language edition published by Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458. Copyright © 2013 Pearson Education, Inc. Russian-language edition copyright © 2014 by ДМК Пресс. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-0-321-90292-4 (англ.)
ISBN 978-5-97060-146-4 (рус.)

Copyright © 2013 Pearson Education, Inc.
© Оформление, перевод на русский язык,
издание, ДМК Пресс, 2015



Положительные отзывы к книге «WebGL: руководство по программированию»

«WebGL – одна из заключительных особенностей, приближающих веб-приложения к обычным, настольным приложениям, и книга «WebGL: программирование трехмерной графики» рассказывает о создании таких веб-приложений. Она охватывает все аспекты, связанные с использованием WebGL – JavaScript, OpenGL ES и базовые приемы программирования графики – помогая получить полное представление обо всем, что вам может потребоваться на начальном этапе. За веб-приложениями – будущее, и эта книга поможет вам попасть туда!»

Дэйв Шрайнер (Dave Shreiner), соавтор «The OpenGL Programming Guide, Eighth Edition»¹; редактор серии «OpenGL Library» (Addison Wesley).

«Стандарт HTML5 превратил Web в высокоэффективную платформу для создания весьма привлекательных интерактивных приложений, способных выполняться в самых разных системах. WebGL является одним из важнейших элементов HTML5, позволяющим веб-программистам использовать всю широту возможностей современных графических ускорителей. Поддержка WebGL разрабатывалась так, чтобы обеспечить максимальную надежность в любых веб-совместимых системах и послужить толчком для появления новых инноваций в разработке трехмерного веб-контента, приложений и пользовательских интерфейсов. Эта книга поможет веб-разработчикам понять, какие удивительные возможности несет в себе новая волна веб-функциональности и использовать их на практике.»

Нейл Треветт (Neil Trevett), вице-президент Mobile Content, NVIDIA; президент The Khronos Group

«Давая ясное описание, сопровождаемое отменными иллюстрациями, эта книга прекрасно справляется с задачей превращения такой сложной темы в простое и практическое руководство. Несмотря на всю сложность WebGL, данная книга яв-

¹ На русский язык было переведено только 4-е издание книги: Д. Шрайнер, М. Ву, Дж. Нейдер, Т. Девис «OpenGL. Руководство по программированию. Библиотека программиста. 4-е изд.», ISBN: 5-94723-827-6, 2006, Питер. – *Прим. перев.*

ляется доступным источником знаний даже для начинающих разработчиков, которым определенно стоит выбрать ее, прежде чем переходить к чему-то другому.»

Эван Барчард (Evan Burchard), автор «Web Game Developer's Cookbook» (Addison Wesley)

«Оба автора имеют большой опыт использования OpenGL. Они сумели распространить его на WebGL, в результате чего получилось отличное руководство, которое может пригодиться не только начинающим, но и опытным специалистам.»

Дэниел Хаэн (Daniel Haehn), программист, детская больница в г. Бостон

«WebGL: руководство по программированию просто и понятно излагает механику создания веб-приложений, воспроизводящих трехмерный графику, без использования массивных библиотек или фреймворков. Отличный источник информации для разработчиков, ищущих вводные руководства по интеграции концепций отображения трехмерный графики с ультрасовременными веб-технологиями.»

Брендон Джонс (Brandon Jones), программист, Google

«Великолепная работа блестящих исследователей. Коичи Мацуда (Koichi Matsuda) уверенно ведет новичков к овладению WebGL, и благодаря его усилиям любой сможет начать использовать эту восхитительную веб-технология, несмотря на ее сложность. Он также включил в книгу описание основных понятий трехмерный графики, заложив фундамент для дальнейшего изучения этой темы. Эта книга станет отличным приобретением для любого веб-дизайнера.»

Крис Маррин (Chris Marrin), редактор спецификации WebGL Spec

«WebGL: руководство по программированию дает отличную возможность пройти путь от новичка до эксперта WebGL. Несмотря на простоту основных понятий, заложенных в WebGL, для работы с этой технологией требуется знание сложного математического аппарата трехмерный моделирования. Книга «WebGL: программирование трехмерной графики» поможет приобрести эти знания и научит применять их на практике. Если вы, в конечном счете, выберете другую библиотеку для работы с трехмерной графикой, отличную от WebGL, знания, полученные при прочтении этой книги, помогут вам быстрее понять, как она действует, и как использовать ее в своих приложениях для решения конкретных задач. И, даже если вы желаете создавать обычные приложения, использующие OpenGL и/или DirectX, «WebGL: программирование трехмерной графики» послужит вам хорошей первой ступенью, потому что большинство книг, посвященных теме программирования трехмерный графики, во многом отстали от современного уровня развития трехмерный технологий. «WebGL: программирование трехмерной графики» поможет вам заложить основы программирования современной трехмерный графики.»

Грегг Таварес (Gregg Tavares), разработчик реализации WebGL в Chrome

*Мудрость приходит с годами, прошлое не возвращается
и время как спираль из полудрагоценных камней...*

— Коичи Мацуда

Моей супруге, семье, друзьям, несущим радость в мою жизнь.

— Роджер Ли



ОГЛАВЛЕНИЕ

Положительные отзывы к книге «WebGL: программирование трехмерной графики»	5
Предисловие	17
Кому адресована эта книга.....	17
О чем рассказывается в этой книге	18
Структура книги	18
Браузеры с поддержкой WebGL.....	22
Примеры программ и сопутствующие ссылки	23
Типографские соглашения	23
Благодарности.....	23
Об авторах	24
Глава 1.	
Обзор WebGL	26
Достоинства WebGL	27
Вы можете заниматься разработкой приложений с трехмерной графикой, используя только текстовый редактор	28
Публикация приложений с трехмерной графикой не вызывает сложностей ..	29
Вы можете использовать всю широту возможностей браузеров	29
Изучение и использование WebGL не вызывает никаких сложностей.....	29
История происхождения WebGL	30
Структура приложений на основе WebGL.....	31
В заключение	32
Глава 2.	
Первые шаги в WebGL	33
Что такое canvas?	33
Использование тега <canvas>	34
DrawRectangle.js.....	36
Самая короткая WebGL-программа: очистка области рисования.....	40
Файл HTML (HelloCanvas.html).....	40
Программа на JavaScript (HelloCanvas.js)	41
Эксперименты с примером программы.....	46
Рисование точки (версия 1)	46
HelloPoint1.html	48
HelloPoint1.js	48
Что такое шейдер?	49
Структура WebGL-программы, использующей шейдеры	51

Инициализация шейдеров	53
Вершинный шейдер	55
Фрагментный шейдер	58
Операция рисования	58
Система координат WebGL	60
Эксперименты с примером программы	61
Рисование точки (версия 2)	62
Использование переменных-атрибутов	63
Пример программы (HelloPoint2.js)	64
Получение ссылки на переменную-атрибут	65
Присваивание значения переменной-атрибуту	66
Семейство методов <code>gl.vertexAttrib3f()</code>	68
Эксперименты с примером программы	70
Рисование точки по щелчку мышью	71
Пример программы (ClickedPoints.js)	72
Регистрация обработчиков событий	73
Обработка события щелчка мышью	75
Эксперименты с примером программы	78
Изменение цвета точки	79
Пример программы (ColoredPoints.js)	80
Uniform-переменные	82
Получение ссылки на uniform-переменную	83
Присваивание значения uniform-переменной	84
Семейство методов <code>gl.uniform4f()</code>	86
В заключение	86

Глава 3.

Рисование и преобразование треугольников..... 88

Рисование множества точек	88
Пример программы (MultiPoint.js)	90
Использование буферных объектов	93
Создание буферного объекта (<code>gl.createBuffer()</code>)	94
Указание типа буферного объекта (<code>gl.bindBuffer()</code>)	95
Запись данных в буферный объект (<code>gl.bufferData()</code>)	96
Типизированные массивы	98
Сохранение ссылки на буферный объект в переменной-атрибуте (<code>gl.vertexAttribPointer()</code>)	99
Разрешение присваивания переменной-атрибуту (<code>gl.enableVertexAttribArray()</code>)	101
Второй и третий параметры метода <code>gl.drawArrays()</code>	102
Эксперименты с примером программы	103
Привет, треугольник	104
Пример программы (HelloTriangle.js)	105
Простые фигуры	106
Эксперименты с примером программы	108
Привет, прямоугольник (HelloQuad)	109
Эксперименты с примером программы	110
Перемещение, вращение и масштабирование	111
Перемещение	112

Пример программы (TranslatedTriangle.js)	113
Вращение	115
Пример программы (RotatedTriangle.js)	117
Матрица преобразования: вращение	121
Матрица преобразования: перемещение	123
И снова матрица вращения	124
Пример программы (RotatedTriangle_Matrix.js)	125
Применение того же подхода для перемещения	128
Матрица преобразования: масштабирование	129
В заключение	130

Глава 4.

Дополнительные преобразования и простая анимация 131

Перемещение с последующим вращением	131
Библиотека матричных преобразований: cuon-matrix.js	132
Пример программы (RotatedTriangle_Matrix4.js)	133
Объединение нескольких преобразований	135
Пример программы (RotatedTranslatedTriangle.js)	137
Эксперименты с примером программы	139
Анимация	140
Основа анимации	141
Пример программы (RotatingTriangle.js)	141
Повторяющиеся вызовы функции рисования (tick())	144
Рисование треугольника после поворота на указанный угол (draw())	145
Запрос на повторный вызов (requestAnimationFrame())	146
Изменение угла поворота (animate())	148
Эксперименты с примером программы	150
В заключение	151

Глава 5.

Цвет и текстура 152

Передача другой информации в вершинные шейдеры	152
Пример программы (MultiAttributeSize.js)	153
Создание нескольких буферных объектов	155
Параметры stride и offset метода gl.vertexAttribPointer()	156
Пример программы (MultiAttributeSize_Interleaved.js)	157
Изменение цвета (varying-переменные)	160
Пример программы (MultiAttributeColor.js)	161
Эксперименты с примером программы	164
Цветной треугольник (ColoredTriangle.js)	165
Сборка и растеризация геометрических фигур	165
Вызовы фрагментного шейдера	169
Эксперименты с примером программы	170
Принцип действия varying-переменных и процесс интерполяции	171
Наложение изображения на прямоугольник	174
Координаты текстуры	176
Пример программы (TexturedQuad.js)	176
Использование координат текстуры (initVertexBuffers())	179
Подготовка и загрузка изображений (initTextures())	179

Подготовка загруженной текстуры к использованию в WebGL	
(loadTexture())	183
Поворот оси Y изображения	183
Выбор текстурного слота (gl.activeTexture())	184
Указание типа объекта текстуры (gl.bindTexture())	185
Настройка параметров объекта текстуры (gl.texParameteri())	187
Присваивание изображения объекту текстуры (gl.texImage2D())	190
Передача текстурного слота фрагментному шейдеру (gl.uniform1i())	192
Передача координат текстуры из вершинного шейдера	
во фрагментный шейдер	193
Извлечение цвета текселя во фрагментном шейдере (texture2D())	193
Эксперименты с примером программы	195
Наложение нескольких текстур на фигуру	196
Пример программы (MultiTexture.js)	197
В заключение	201

Глава 6.

Язык шейдеров OpenGL ES (GLSL ES)	203
Краткое повторение основ шейдеров	203
Обзор GLSL ES	204
Привет, шейдер!	205
Основы	205
Порядок выполнения	205
Комментарии	205
Данные (числовые и логические значения)	206
Переменные	206
GLSL ES – типизированный язык	207
Простые типы	207
Присваивание и преобразования типов	208
Операции	209
Векторы и матрицы	210
Присваивание и конструирование	211
Доступ к компонентам	213
Операции	216
Структуры	218
Присваивание и конструирование	219
Доступ к членам	219
Операции	219
Массивы	220
Семплы	221
Приоритеты операторов	221
Условные операторы и циклы	222
Инструкции if и if-else	222
Инструкция for	223
Инструкции continue, break, discard	223
Функции	224
Объявления прототипов	225
Квалификаторы параметров	226
Встроенные функции	227

Глобальные и локальные переменные	227
Квалификаторы класса хранения	228
Квалификатор const	228
uniform-переменные	230
varying-переменные	230
Квалификаторы точности	230
Директивы препроцессора	233
В заключение	235

Глава 7.

Вперед, в трехмерный мир	236
Что хорошо для треугольников, хорошо и для кубов	236
Определение направления взгляда	237
Точка наблюдения, точка направления взгляда и направление вверх	238
Пример программы (LookAtTriangles.js)	240
Сравнение LookAtTriangles.js с RotatedTriangle_Matrix4.js	243
Взгляд на повернутый треугольник с указанной позиции	245
Пример программы (LookAtRotatedTriangles.js)	246
Эксперименты с примером программы	247
Изменение точки наблюдения с клавиатуры	249
Пример программы (LookAtTrianglesWithKeys.js)	249
Недостающие части	252
Определение видимого объема в форме прямоугольного параллелепипеда	252
Определение видимого объема	253
Определение границ видимого объема в форме параллелепипеда	254
Пример программы (OrthoView.html)	256
Пример программы (OrthoView.js)	257
Изменение содержимого HTML-элемента из JavaScript	258
Вершинный шейдер	259
Изменение near или far	260
Восстановление отсеченных частей треугольников	
(LookAtTrianglesWithKeys_ViewVolume.js)	262
Эксперименты с примером программы	264
Определение видимого объема в форме четырехгранной пирамиды	265
Определение границ видимого объема в форме четырехгранной	
пирамиды	267
Пример программы (PerspectiveView.js)	269
Назначение матрицы проекции	271
Использование всех матриц (модели, вида и проекции)	272
Пример программы (PerspectiveView_mvp.js)	274
Эксперименты с примером программы	276
Правильная обработка объектов переднего и заднего плана	277
Удаление скрытых поверхностей	280
Пример программы (DepthBuffer.js)	282
Z-конфликт	283
Привет, куб	285
Рисование объектов с использованием индексов и координат вершин	287
Пример программы (HelloCube.js)	288

Запись координат вершин, цветов и индексов в буферный объект	291
Добавление цвета для каждой грани куба	293
Пример программы (ColoredCube.js)	295
Эксперименты с примером программы	296
В заключение	297

Глава 8.

Освещение объектов	299
Освещение трехмерных объектов	299
Типы источников света	300
Типы отраженного света	302
Затенение при направленном освещении в модели диффузного отражения	304
Использование направления света и ориентации поверхности в модели диффузного отражения	305
Ориентация поверхности: что такое нормаль?	307
Пример программы (LightedCube.js)	309
Добавление затенения, обусловленного фоновым освещением	315
Пример программы (LightedCube_ambient.js)	316
Освещенность перемещаемого и вращаемого объекта	317
Волшебство матриц: транспонированная обратная матрица	319
Пример программы (LightedTranslatedRotatedCube.js)	320
Освещение точечным источником света	322
Пример программы (PointLightedCube.js)	323
Более реалистичное затенение: вычисление цвета для каждого фрагмента	326
Пример программы (PointLightedCube_perFragment.js)	327
В заключение	328

Глава 9.

Иерархические объекты	329
Рисование составных объектов и управление ими	329
Иерархическая структура	331
Модель с единственным сочленением	332
Пример программы (JointModel.js)	333
Рисование иерархической структуры (draw())	337
Модель со множеством сочленений	339
Пример программы (MultiJointModel.js)	340
Рисование сегментов (drawBox())	343
Рисование сегментов (drawSegment())	345
Шейдер и объект программы: роль initShaders()	349
Создание объектов шейдеров (gl.createShader())	350
Сохранение исходного кода шейдеров в объектах шейдеров (g.shaderSource())	351
Компиляция объектов шейдеров (gl.compileShader())	351
Создание объекта программы (gl.createProgram())	353
Подключение объектов шейдеров к объекту программы (gl.attachShader())	354
Компоновка объекта программы (gl.linkProgram())	355

Сообщение системе WebGL о готовности объекта программы (gl.useProgram())	356
Реализация initShaders()	357
В заключение	359

Глава 10.

Продвинутые приемы	360
Вращение объекта мышью	360
Как реализовать вращение объекта	361
Пример программы (RotateObject.js)	361
Выбор объекта	363
Как реализовать выбор объекта	364
Пример программы (PickObject.js)	365
Выбор грани объекта	368
Пример программы (PickFace.js)	368
Эффект индикации на лобовом стекле (ИЛС)	371
Как реализовать ИЛС	371
Пример программы (HUD.html)	372
Пример программы (HUD.js)	373
Отображение трехмерного объекта в веб-странице (3DoverWeb)	375
Туман (атмосферный эффект)	376
Реализация эффекта тумана	376
Пример программы (Fog.js)	377
Использование значения w (Fog_w.js)	379
Создание круглой точки	380
Как нарисовать круглую точку	380
Пример программы (RoundedPoints.js)	382
Альфа-смешивание	383
Как реализовать альфа-смешивание	383
Пример программы (LookAtBlendedTriangles.js)	384
Как должна действовать функция смешивания	385
Альфа-смешивание для трехмерных объектов (BlendedCube.js)	386
Рисование при наличии прозрачных и непрозрачных объектов	388
Переключение шейдеров	389
Как реализовать переключение шейдеров	390
Пример программы (ProgramObject.js)	390
Использование нарисованного изображения в качестве текстуры	394
Объект буфера кадра и объект буфера отображения	395
Как реализовать использование нарисованного объекта в качестве текстуры	397
Пример программы (FramebufferObject.js)	398
Создание объекта буфера кадра (gl.createFramebuffer())	399
Создание объекта текстуры, настройка его размеров и параметров	400
Создание объекта буфера отображения (gl.createRenderbuffer())	401
Связывание объекта буфера отображения с типом и настройка его размера (gl.bindRenderbuffer(), gl.renderbufferStorage())	401
Подключение объекта текстуры, как ссылки на буфер цвета (gl.bindFramebuffer(), gl.framebufferTexture2D())	403

Подключение объекта буфера отображения к объекту буфера кадра (gl.framebufferRenderbuffer())	404
Проверка корректности настройки объекта буфера кадра (gl.checkFramebufferStatus())	405
Рисование с использованием объекта буфера кадра	406
Отображение теней	407
Как реализуются тени	408
Пример программы (Shadow.js)	409
Увеличение точности	414
Пример программы (Shadow_highp.js)	415
Загрузка и отображение трехмерных моделей	417
Формат OBJ	419
Формат файла MTL	420
Пример программы (OBJViewer.js)	421
Объект, определяемый пользователем	424
Пример программы (реализация анализа содержимого файла)	425
Обработка ситуации потери контекста	432
Как реализовать обслуживание ситуации потери контекста	433
Пример программы (RotatingTriangle_contextLost.js)	434
В заключение	436

Приложение А.

В WebGL нет необходимости использовать рабочий и фоновый буферы	438
--	------------

Приложение В.

Встроенные функции в языке GLSL ES 1.0	441
Функции для работы с угловыми величинами и тригонометрические функции	441
Экспоненциальные функции	442
Общие функции	443
Геометрические функции	445
Матричные функции	446
Векторные функции	447
Функции для работы с текстурами	448

Приложение С.

Матрицы проекций	449
Матрица ортогональной проекции	449
Матрица перспективной проекции	449

Приложение D.

WebGL/OpenGL: лево- или правосторонняя система координат? ...	450
Пример программы CoordinateSystem.js	451
Удаление скрытых поверхностей и усеченная система координат	453
Усеченная система координат и видимый объем	454
Теперь все правильно?	456
В заключение	459

Приложение Е.	
Транспонированная обратная матрица	460
Приложение F.	
Загрузка шейдеров из файлов	464
Приложение G.	
Мировая и локальная системы координат	466
Локальная система координат.....	466
Мировая система координат	467
Преобразования и системы координат.....	469
Приложение H.	
Настройка поддержки WebGL в веб-браузере	470
Словарь терминов	472
Список литературы	477
Предметный указатель	478



ПРЕДИСЛОВИЕ

WebGL – это технология рисования, отображения и интерактивного взаимодействия с трехмерной компьютерной графикой в веб-браузерах. Традиционно поддержка трехмерной графики ограничивалась высокопроизводительными компьютерами или специализированными игровыми консолями, а ее программирование требовало применения сложных алгоритмов. Однако, благодаря росту производительности персональных компьютеров и расширению возможностей браузеров стало возможным создание и отображение трехмерной графики с применением веб-технологий. Эта книга представляет собой исчерпывающий обзор возможностей WebGL и постепенно, шаг за шагом знакомит читателя с основами создания приложений WebGL. В отличие от других технологий для работы с трехмерной графикой, таких как OpenGL и Direct3D, WebGL предназначена для использования в веб-страницах и не требует установки специализированных расширений или библиотек. Благодаря этому можно сразу опробовать примеры программ, имея лишь стандартное окружение. А так как все основано на веб-технологиях, вновь созданные программы легко можно публиковать в Сети. Одно из преимуществ WebGL заключается в том, что приложения конструируются как веб-страницы, то есть одна и та же программа успешно будет выполняться на самых разных устройствах, таких как смартфоны, планшетные компьютеры и игровые консоли. Это означает, что WebGL будет оказывать все более усиливающееся влияние на сообщество разработчиков и станет одним из основных инструментов программирования графики.

Кому адресована эта книга

Мы писали эту книгу для специалистов двух основных категорий: веб-разработчиков, ищущих возможность добавления трехмерной графики в свои веб-страницы и приложения, и программистов, занимающихся проблемами реализации трехмерной графики, желающих понять, как применить свои знания в веб-окружении.

Для специалистов из первой категории – веб-разработчиков, хорошо знакомых со стандартными веб-технологиями, такими как HTML и JavaScript, и стремящихся внедрить трехмерную графику в свои веб-страницы или веб-приложения, – WebGL является простым и мощным решением. Эту технологию с успехом можно применять для добавления трехмерной графики в веб-страницы, с целью повысить качество пользовательского интерфейса веб-приложений, и даже для разработки сложных графических и игровых приложений, выполняющихся в веб-браузерах.

Специалистам из второй категории – программистам, имеющим опыт использования основных прикладных программных интерфейсов (API) создания трехмерной графики, таких как Direct3D или OpenGL, и интересующимся возможностью применения своих знаний в веб-окружении, – будет любопытно поближе познакомиться с возможностью создания сложных приложений для работы с трехмерными изображениями, выполняющимися в современных веб-браузерах.

Однако, мы постарались сделать книгу доступной для широкого круга читателей и использовали пошаговый подход к знакомству с особенностями WebGL, не предполагая наличия у читателей опыта программирования двух- или трехмерной графики. Мы полагаем, что она будет интересна также:

- обычным программистам, кому будет любопытно узнать, как идет развитие веб-технологий в направлении поддержки графических возможностей;
- студентам, изучающим приемы программирования двух- и трехмерной графики, потому что WebGL позволяет легко экспериментировать с графикой в веб-браузере и не требует установки и настройки полноценной среды программирования;
- веб-разработчикам, исследующим ультрасовременные возможности, поддерживаемые мобильными устройствами с новейшими версиями мобильных веб-браузеров.

О чем рассказывается в этой книге

Эта книга охватывает WebGL 1.0 API, наряду с соответствующими функциями JavaScript. Здесь вы узнаете, как связаны между собой HTML, JavaScript и WebGL, как устанавливать и запускать приложения WebGL и как программировать сложные трехмерные «шейдеры» на JavaScript. Книга подробно рассказывает, как описывать вершинные и фрагментные шейдеры, как реализовать разные усовершенствованные приемы отображения, такие как попиксельное освещение или затенение, а также об основных приемах взаимодействий, такими как выделение трехмерных объектов. В каждой главе создается несколько действующих приложений, на примере которых демонстрируются ключевые особенности WebGL. Закончив читать эту книгу, вы будете готовы писать приложения с применением WebGL, использующие всю мощь веб-браузеров и аппаратной поддержки графики.

Структура книги

Эта книга организована в виде последовательного описания WebGL API и связанных с ним функций, чтобы вы могли выстраивать свое здание знаний WebGL постепенно.

Глава 1 – Обзор WebGL

В этой главе дается короткое представление WebGL, отмечаются некоторые ключевые особенности и преимущества, а также дается история происхождения.

Завершается глава разъяснением взаимоотношений между WebGL, HTML5 и JavaScript, и как использовать веб-браузеры для исследования WebGL.

Глава 2 – Первые шаги в WebGL

В этой главе описывается элемент `<canvas>` и основные возможности WebGL на примере нескольких простых программ. Все программы написаны на JavaScript и используют WebGL для отображения и взаимодействий с простыми геометрическими фигурами на веб-странице. Примеры программ подчеркивают следующие важные моменты: (1) как WebGL использует элемент `<canvas>` и как рисовать на его поверхности; (2) как посредством JavaScript осуществляется связь между HTML и WebGL; (3) работа с простыми функциями рисования WebGL; и (4) роль шейдеров в WebGL.

Глава 3 – Рисование и преобразование треугольников

Эта глава закладывает фундамент для дальнейших исследований особенностей рисования более сложных фигур и манипулирования этими фигурами в трехмерном пространстве. В этой главе рассматриваются: (1) критически важная роль треугольников в трехмерной графике и поддержка рисования треугольников в WebGL; (2) как с помощью множества треугольников рисуются другие простые фигуры; (3) простые преобразования, такие как перемещение, вращение и масштабирование с применением простых уравнений; и (4) как матричные операции упрощают преобразования.

Глава 4 – Дополнительные преобразования и простая анимация

В этой главе вы продолжите исследование преобразований и начнете объединять преобразования в анимации. Здесь вы: (1) познакомитесь с библиотекой матричных преобразований, скрывающей математические тонкости операций с матрицами; (2) научитесь пользоваться библиотекой, позволяющей просто и быстро комбинировать преобразования; и (3) исследуете анимационные эффекты и библиотеку, помогающую организовать анимацию с участием простых фигур. Приемы, представленные в этой главе, образуют основу для конструирования гораздо более сложных программ WebGL и будут использоваться в примерах программ в последующих главах.

Глава 5 – Цвет и текстура

Опираясь на фундамент, заложенный в предыдущих главах, вы более детально познакомитесь с WebGL, исследовав в этой главе следующие три темы: (1) как, помимо координат вершин, передать в вершинный шейдер дополнительную информацию, например, цвет; (2) преобразование фигуры во фрагмент, то есть процедура перехода от вершинного шейдера к фрагментному шейдеру, известная как процесс растеризации; и (3) как «натягивать» изображения (или текстуры) на поверхности фигур или объектов. Эта глава является завершающей в исследовании ключевых особенностей WebGL.

Глава 6 – Язык шейдеров OpenGL ES (GLSL ES)

В этой главе мы оторвемся от изучения примеров программ WebGL и исследуем основные особенности OpenGL ES Shading Language (GLSL ES) – языка программирования шейдеров. Здесь рассказывается о: (1) данных, переменных и типах переменных; (2) векторах, матрицах, структурах, массивах и образцах; (3) операторах, инструкциях управления потоком выполнения и функциях; (4) атрибутах, uniform-переменных и varying-переменных; (5) квалификаторах точности; и (6) препроцессоре и директивах. К концу этой главы вы будете понимать язык GLSL ES и знать, как его использовать для создания различных шейдеров.

Глава 7 – Вперед, в трехмерный мир

Эта глава является первым шагом в трехмерный мир и исследует последствия перехода из двухмерного мира в трехмерный. В частности, вы исследуете: (1) представление точки зрения пользователя в трехмерном мире; (2) как управлять объемом трехмерного пространства; (3) отсечение, или клиппинг (clipping); (4) объекты переднего и заднего плана; и (5) рисование трехмерных объектов – куб. Все эти проблемы оказывают большое влияние на рисование трехмерных сцен и их отображение. Имеющие решать их совершенно необходимо для создания неотразимых трехмерных сцен.

Глава 8 – Освещение объектов

Основное внимание в этой главе уделяется освещению объектов, исследованию разных источников освещения и их влияния на трехмерные сцены. Освещение играет важную роль в придании реалистичности трехмерным сценам, потому что правильно подобранное освещение придает ощущение глубины.

В этой главе обсуждаются следующие ключевые моменты: (1) тени, затенение и разнотипные источники света, включая точечные, направленные и рассеянные; (2) отражение света в трехмерных сценах и два основных типа отражения: рассеянное и фоновое; и (3) тонкости затенения и как реализовать эффект освещения, чтобы объекты выглядели трехмерными.

Глава 9 – Иерархические объекты

Эта глава завершает описание основных особенностей и приемов программирования с применением WebGL. К концу этой главы вы овладеете всеми основами WebGL и будете в состоянии самостоятельно создавать реалистичные и интерактивные трехмерные сцены. Основное внимание в этой главе уделяется иерархическим объектам, играющим важную роль, потому что позволяют перейти от простых объектов, таких как кубы и блоки, к более сложным конструкциям, которые можно использовать в роли персонажей игр, роботов и даже моделировать на их основе людей.

Глава 10 – Продвинутые приемы

В этой главе затрагиваются различные важнейшие приемы, основанные на приемах, изучаемых в предыдущих главах и позволяющие создавать интерактивные, реалистичные трехмерные изображения. Описания всех приемов сопровождаются

ся законченными примерами, которые вы можете использовать при разработке своих приложений WebGL.

Приложение А – В WebGL нет необходимости использовать рабочий и фоновый буферы

В этом приложении рассказывается, почему программы WebGL не нуждаются в перестановке рабочего и фонового буферов.

Приложение В – Встроенные функции в языке GLSL ES 1.0

Это приложение – справочник по всем встроенным функциям в языке OpenGL ES Shading Language.

Приложение С – Матрицы проекций

В этом приложении представлены матрицы проекций, сгенерированные функциями `Matrix4.setOrtho()` и `Matrix4.setPerspective()`.

Приложение D – WebGL/OpenGL: лево- или правосторонняя система координат?

В этом приложении рассказывается, как WebGL и OpenGL работают с системой координат. Здесь же поясняется, что обе технологии, WebGL и OpenGL, нейтральны к выбору системы координат.

Приложение Е – Транспонированная обратная матрица

В этом приложении разъясняется, как с помощью операций обращения и транспонирования матрицы модели можно выполнять преобразования векторов нормали.

Приложение F – Загрузка шейдеров из файлов

В этом приложении рассказывается, как организовать загрузку программ шейдеров из файлов.

Приложение G – Мировая и локальная системы координат

В этом приложении описываются разные системы координат и их применение в трехмерной графике.

Приложение H – Настройка поддержки WebGL в веб-браузере

В этом приложении объясняется, как с помощью дополнительных настроек веб-браузера гарантировать правильное отображение графики, сгенерированной средствами WebGL.

Браузеры с поддержкой WebGL

На момент написания этих строк, технология WebGL поддерживалась браузерами: Chrome, Firefox, Safari и Opera. Печально, что некоторые распространенные браузеры, такие как IE9 (Microsoft Internet Explorer), не имеют такой поддержки. В процессе работы над этой книгой мы пользовались браузером Chrome, выпущенным компанией Google, который помимо WebGL поддерживает еще массу полезных возможностей, таких как консоль для отладки функций. Примеры программ, что приводятся в этой книге, опробованы в следующих окружениях (см. табл. П.1), но мы полагаем, что они будут работать в любых браузерах с поддержкой WebGL.

Таблица П.1. Окружение, в которых опробовались примеры программ

Браузер	Chrome (25.0.1364.152 m)
ОС	Windows 7 и 8
Графическая карта	NVIDIA Quadro FX 380, NVIDIA GT X 580, NVIDIA GeForce GTS 450, Mobile Intel 4 Series Express Chipset Family, AMD Radeon HD 6970

На странице www.khronos.org/webgl/wiki/BlacklistsAndWhitelists можно найти постоянно обновляемый список графических карт, имеющих известные проблемы.

Чтобы проверить, поддерживает ли ваш браузер WebGL, запустите его и перейдите на веб-сайт этой книги: <https://sites.google.com/site/webglbook/>.

Перейдите по ссылке **Chapter 3** и выберите пример `HelloTriangle` в списке. Если вы увидите красный треугольник, как показано на рис. П.1, значит ваш браузер поддерживает WebGL.

Если ваш браузер не смог вывести красный треугольник, загляните в приложение Н, где описывается, как изменить настройки браузера, чтобы активизировать в нем поддержку WebGL.

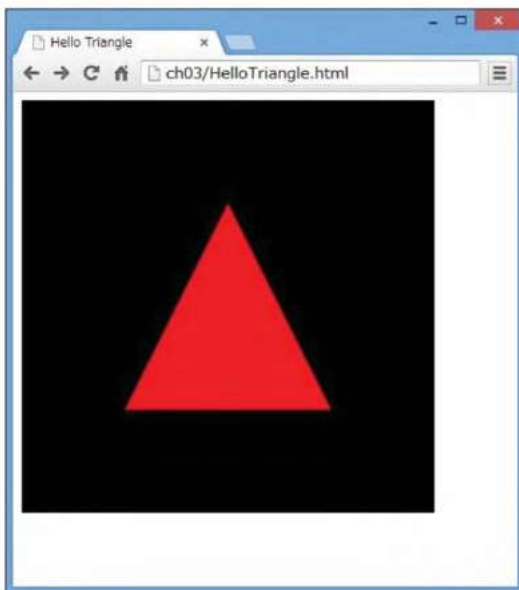


Рис. П.1. При выборе примера `HelloTriangle` должен появиться красный треугольник

Примеры программ и сопутствующие ссылки

Все примеры программ и сопутствующие ссылки, что приводятся в этой книге, доступны на вспомогательных веб-сайтах. Официальный сайт книги находится на сайте издательства: www.informit.com/title/9780321902924. Авторы поддерживают свой сайт книги: <https://sites.google.com/site/webglbook/>.

Последний из этих сайтов содержит ссылки на все примеры программ, что приводятся в этой книге. Вы можете опробовать каждый из них, просто щелкая на ссылках.

Если вы пожелаете внести свои изменения в программы, загрузите zip-файл со всеми примерами, доступный на втором сайте, на свой локальный диск. Обратите внимание, что каждый пример состоит из двух файлов – файла HTML и соответствующего ему файла JavaScript – в одной папке. Например, пример программы HelloTriangle состоит из двух файлов: HelloTriangle.html и HelloTriangle.js. Чтобы запустить программу HelloTriangle, выполните двойной щелчок на файле HelloTriangle.html.

Типографские соглашения

В этой книге используются следующие типографские соглашения:

- **Жирный шрифт** – используется для выделения ключевых терминов и важных слов.
- *Курсив* – используется для выделения имен параметров и ссылок.
- Моноширинный шрифт – используется для оформления примеров кода; имен методов, функций, переменных; параметров команд; имен объектов JavaScript; имен файлов; тегов HTML.

Благодарности

Нам было приятно получить помощь от многих талантливых личностей в процессе работы над этой книгой – и над начальной японской версией, и над последующей за ней английской версией.

Такафуми Канда (Takafumi Kanda) помог нам с многочисленными фрагментами кода для нашей библиотеки поддержки и примеров программ, без его участия мы просто не смогли бы закончить эту книгу. Ясуко Кикучи (Yasuko Kikuchi), Чи Онума (Chie Onuma) и Ючи Нишизава (Yuichi Nishizawa) оказали неоценимую помощь при работе над ранними версиями книги. Хочется заметить, что один глубокомысленный комментарий госпожи Кикучи буквально остановил работу над книгой и заставил нас полностью переписать несколько разделов, чтобы обеспечить большую техническую точность. Хироюки Танака (Hiroyuki Tanaka) и Казухира Оониши (Kazuhira Oonishi) (iLinx) оказали большую помощь с примера-

ми программ, а Тэрухиса Камачи (Teruhisa Kamachi) и Тетсуо Юшитани (Tetsuo Yoshitani) помогли в работе над разделами, посвященными HTML5 и JavaScript. Члены рабочей группы WebGL, особенно Кен Рассел (Ken Russell) (Google), Крис Марин (Chris Marin) (Apple) и Дэн Гинзбург (Dan Ginsburg) (AMD) любезно согласились ответить на массу технических вопросов. Мы получили привилегию воспользоваться поддержкой президента Khronos Group, Нейла Треветта (Neil Trevett), и весьма благодарны Хитоши Касаи (Hitoshi Kasai) (руководителю MIACIS Associates) за то, что связал нас с мистером Треветтом и членами рабочей группы WebGL. Мы также хотим выразить благодарность Ксавье Мишелю (Xavier Michel) и Макото Сато (Makoto Sato) (Софийский университет, Токио), оказавшим неоценимую помощь в переводе оригинального текста и решении проблем, возникавшим во время перевода. Английская версия книги тщательно была выверена Джеффом Гилбертом (Jeff Gilbert), Риком Рафии (Rick Rafey) и Даниэлем Ханом (Daniel Haehn), которые дали ценные технические замечания, позволившие значительно улучшить эту книгу. Кроме того, мы хотим выразить благодарность Лауре Левин (Laura Lewin) и Оливии Бесижио (Olivia Basegio) из издательства Pearson, помогавшим в организации публикации книги и обеспечивавшим неуклонное движение вперед.

Мы оба выражаем признательность авторам книг «Red Book» (OpenGL Programming Guide) и «Gold Book» (OpenGL ES 2.0 Programming Guide), выпущенных издательством Pearson, без которых эта книга была бы невозможной. Мы надеемся, что эта книга в какой-то степени послужит выражением нашей признательности.

Об авторах

Доктор Коичи Мацуда (Kouichi Matsuda) имеет большой опыт в области разработки пользовательских интерфейсов и новых мультимедийных программных продуктов. Много лет занимался разработкой продуктов и исследованиями в таких компаниях, как NEC, Sony Corporate Research и Sony Computer Science Laboratories. В настоящее время занимает должность руководителя отдельной исследовательской группы, занимающейся изысканиями в области пользовательских интерфейсов и взаимодействий человека с машиной. В свое время занимался проектированием трехмерного виртуального мира «PAW» (Personal Agent-oriented virtual World – личный агент-ориентированный виртуальный мир), с самого начала был вовлечен в разработку стандарта VRML97 (ISO/IEC 14772-1:1997)¹, и продолжает участвовать в жизни обоих сообществ – VRML и X3D (предшественник WebGL). Является автором 15 книг по компьютерным технологиям. Перевел 25 книг на японский язык. Считается опытным экспертом в области пользовательских интерфейсов, взаимодействий человека с машиной, обработки естественных языков, построения развлекательных сетевых служб и интерфейсов агентных систем. Всегда следит за появлением новых возможностей в области

¹ VRML (Virtual Reality Modeling Language – язык моделирования виртуальной реальности). – *Прим. перев.*

технологий. Помимо профессиональной деятельности любит посещать горячие термальные источники, отдыхать на море летом, увлекается виноделием, а также рисованием иллюстраций к книгам издательства MANGA. Защитил докторскую диссертацию в Токийском университете. Связаться с ним можно по адресу: WebGL.prog.guide@gmail.com.

Доктор Роджер Ли (Rodger Lea) – адъюнкт-профессор в центре Media and Graphics Interdisciplinary Centre, в Университете Британской Колумбии (University of British Columbia), с интересом в области мультимедиа и распределенных вычислений. Имеет более чем 20-летний опыт руководства исследовательской группой в академической и промышленной среде, занимался разработкой ранних версий трехмерных миров, участвовал в работе над стандартом VRML97, разрабатывал мультимедийные операционные системы, участвовал в создании прототипов интерактивного цифрового телевидения и в разработке стандартов домашних мультимедийных сетей. Опубликовал более 60 статей и написал 3 книги, является держателем 12 патентов. В настоящее время занимается исследованием растущего «Интернета вещей» (Internet of Things, IoT)², но сохраняет влечение ко всему, что связано с графикой и мультимедиа.

² Концепция вычислительной сети физических объектов («вещей»), оснащённых встроенными технологиями для взаимодействия друг с другом или с внешней средой. – *Прим. перев.*

ГЛАВА 1.

Обзор WebGL

WebGL – это технология, позволяющая рисовать, отображать и взаимодействовать со сложной, интерактивной трехмерной компьютерной графикой в веб-браузерах. Традиционно поддержка трехмерной графики ограничивалась высокопроизводительными компьютерами или специализированными игровыми консолями, а ее программирование требовало применения сложных алгоритмов. Однако, благодаря росту производительности персональных компьютеров и расширению возможностей браузеров стало возможным создание и отображение трехмерной графики с применением веб-технологий. WebGL, в сочетании с HTML5 и JavaScript, делает трехмерную графику доступной для веб-разработчиков и открывает возможность создания веб-приложений следующего поколения, с простыми и понятными пользовательскими интерфейсами и веб-контентом. Некоторые примеры таких интерфейсов показаны на рис. 1.1. В ближайшие годы можно ожидать появления программ, использующих WebGL, на самых разных устройствах, от стандартных ПК до бытовой электроники, смартфонов и планшетных компьютеров.



Рис. 1.1. Сложная трехмерная графика в браузере. © 2011 Хиромаса Хори (Hiromasa Horie) (слева), © 2012 Коичи Мацуда (Kouichi Matsuda) (справа)

HTML5, последняя версия стандарта HTML, дополнила традиционный HTML средствами для работы с 2-мерной графикой, сетевых взаимодействий и доступа к локальному хранилищу. С появлением HTML5 браузеры быстро развились из простых инструментов отображения в сложные платформы для приложений. Однако с развитием появилась потребность в интерфейсах и графических возможностях, не ограничивающихся двумя измерениями. По этой причине была создана технология WebGL, занявшая центральное место в создании слоя отображения для новых приложений с трехмерной графикой, выполняющихся под управлением браузеров.

Традиционно для воспроизведения трехмерной графики требовалось создавать автономные приложения на таких языках программирования, как C или C++, использующие мощные графические библиотеки, такие как OpenGL и Direct3D. Однако, с появлением WebGL трехмерную графику стало возможным воспроизводить на стандартных веб-страницах, используя знакомые языки HTML и JavaScript – с небольшим объемом кода.

Особенно важно, что WebGL поддерживается браузерами, как технология по умолчанию для отображения трехмерной графики, что дает возможность использовать ее непосредственно, без необходимости устанавливать специальные расширения или библиотеки. Но самое замечательное, что основой всего этого является браузер, то есть вы можете запускать приложения WebGL на самых разных платформах, от быстродействующих ПК до бытовой электроники, планшетных компьютеров и смартфонов.

Эта глава познакомит вас с технологией WebGL в общих чертах, обозначит ключевые ее особенности и достоинства, а так же расскажет о происхождении WebGL. Кроме того, здесь вы узнаете, как связаны между собой WebGL, HTML5 и JavaScript, и познакомитесь со структурой программ, использующих WebGL.

Достоинства WebGL

С развитием HTML, веб-разработчики получали новые возможности для создания все более и более сложных веб-приложений. Первоначально язык HTML предлагал только возможность отображения статического контента, но с добавлением поддержки языка сценариев JavaScript появилась возможность реализации более сложных взаимодействий и отображения динамического контента. Стандарт HTML5 добавил дополнительные возможности, включая поддержку 2-мерной графики в виде тега `canvas`. Это дало возможность размещать на страницах разные графические элементы, от танцующих мультипликационных персонажей до вывода анимации, соответствующей вводу пользователя.

Создание технологии WebGL стало еще одним шагом вперед, позволившим отображать и манипулировать трехмерной графикой на веб-страницах с помощью JavaScript. Благодаря WebGL, разработчики могут создавать продвинутые пользовательские интерфейсы, трехмерные игры и использовать трехмерную графику для визуализации различной информации из Интернета. Несмотря на внушительные возможности, WebGL отличается от других технологий доступностью и простотой в использовании, что способствует ее быстрому распространению. В частности:

- вы можете заниматься разработкой приложений с трехмерной графикой, используя только текстовый редактор и браузер;
- вы легко можете опубликовать свое приложение с трехмерной графикой, используя стандартные веб-технологии, и сделать его доступным для своих друзей и других пользователей Сети;
- вы можете использовать всю широту возможностей браузеров;
- изучение и использование WebGL не вызывает никаких сложностей, потому что имеется огромное количество обучающих материалов и примеров программного кода.

Вы можете заниматься разработкой приложений с трехмерной графикой, используя только текстовый редактор

Одной из замечательных особенностей разработки приложений с использованием WebGL является отсутствие необходимости в установке среды разработки. Как уже говорилось выше, так как WebGL является встроенной технологией, для разработки приложений с трехмерной графикой не требуется устанавливать специализированные инструменты, такие как компиляторы и компоновщики. Чтобы опробовать примеры программ, которые приводятся в этой книге, нужен всего лишь браузер с поддержкой WebGL. Если вы захотите поэкспериментировать с этими приложениями или написать свое, достаточно иметь стандартный текстовый редактор (например, Notepad или TextEdit). На рис. 1.2 демонстрируется приложение WebGL, выполняющееся в браузере Chrome, и файл HTML, открытый в редакторе Notepad. Файл с разметкой HTML загружает сценарий на языке JavaScript (RotateObject.js), использующий возможности технологии WebGL, который так же можно редактировать в простом текстовом редакторе.



Браузер (Chrome)



Notepad

Рис. 1.2. Для разработки WebGL-приложений с трехмерной графикой необходимы только эти инструменты

Публикация приложений с трехмерной графикой не вызывает сложностей

Традиционно, приложения с трехмерной графикой разрабатывались на таких языках, как С или С++ и компилировались в выполняемые файлы для конкретных платформ. Это означает, например, что версия для Macintosh не будет работать в Windows или Linux. Кроме того, пользователям требовалось установить не только само приложение, но еще и дополнительные библиотеки, что только добавляло сложностей для тех, кто стремился поделиться своим приложением.

Приложения на основе WebGL, напротив, благодаря тому, что состоят из файлов HTML и JavaScript, легко могут распространяться, так как для этого достаточно всего лишь разместить их на веб-сервере, как обычные веб-страницы, или отправить файлы HTML и JavaScript по электронной почте. Например, на рис. 1.3 показаны примеры WebGL-приложений, опубликованных компанией Google и доступных по адресу: <http://code.google.com/p/webglsamples/>.



Рис. 1.3. Примеры WebGL-приложений, опубликованных компанией Google (с разрешения автора, Грегга Тавареса (Gregg Tavares), Google)

Вы можете использовать всю широту возможностей браузеров

Поскольку WebGL-приложения фактически являются веб-страницами, появляется возможность использовать всю широту возможностей веб-браузеров, таких как добавление кнопок, вывод диалогов и текста, проигрывание аудио- и видеороликов, а также обмен данными с веб-серверами. Все эти дополнительные особенности доступны изначально, тогда как в традиционных приложениях с трехмерной графикой их необходимо реализовывать явно.

Изучение и использование WebGL не вызывает никаких сложностей

Спецификация WebGL основана на открытом стандарте OpenGL, который уже много лет используется в качестве основы для разработки графических приложений, видеониг и систем автоматизированного проектирования. Название «WebGL» можно интерпретировать как «OpenGL для веб-браузеров». Так как

технология OpenGL используется на самых разных платформах уже более 20 лет, и за эти годы было написано огромное количество учебников, справочников, статей и примеров программ, использующих ее, которыми с успехом можно использовать и при изучении WebGL.

История происхождения WebGL

Из технологий отображения трехмерной графики на персональных компьютерах наибольшее распространение получили Direct3D и OpenGL. Direct3D – составная часть пакета технологий Microsoft DirectX – это технология отображения трехмерной графики, предназначенная для использования на платформе Windows. Она является лицензионным программным интерфейсом (API) и контролируется корпорацией Microsoft. Альтернативная ей технология OpenGL получила гораздо более широкое распространение, благодаря ее открытости. Реализации OpenGL доступны для Macintosh, Linux и различных устройств, таких как смартфоны, планшетные компьютеры и игровые консоли (PlayStation и Nintendo). Имеются также реализации для Windows, которые представляют прямые альтернативы Direct3D.

Первоначально технология OpenGL была разработана в компании Silicon Graphics Inc. и опубликована как открытый стандарт в 1992 году. С тех пор вышло несколько версий стандарта OpenGL, оказавшего большое влияние на развитие трехмерной графики, создание программных продуктов и даже киноиндустрию. Последней версией OpenGL для настольных ПК на момент написания этих строк была версия 4.3. Несмотря на то, что технология WebGL корнями уходит в OpenGL, в действительности она является дальнейшим развитием версии OpenGL для встраиваемых систем, таких как смартфоны и игровые консоли. Эта версия, известная как OpenGL ES (for Embedded Systems – для встраиваемых систем), создана в 2003–2004 годах и затем была обновлена в 2007 году (ES 2.0) и в 2012 (ES 3.0). WebGL основана на версии ES 2.0. В последние годы быстро растет число устройств и процессоров, поддерживающих спецификацию, в число которых входят смартфоны (iPhone и Android), планшетные компьютеры и игровые консоли. Отчасти такой успех обусловлен добавлением в OpenGL ES новых особенностей, а также удалением множества устаревших и ненужных функций, что сделало спецификацию более легковесной и при этом достаточной для создания выразительной и привлекательной трехмерной графики.

На рис. 1.4 показано, как связаны между собой OpenGL, OpenGL ES 1.1/2.0/3.0 и WebGL. Так как сама спецификация OpenGL продолжала непрерывное развитие от версии 1.5 к версиям 2.0 ... 4.3, спецификация OpenGL ES была стандартизована как подмножество определенных версий OpenGL (OpenGL 1.5 и OpenGL 2.0).

Как показано на рис. 1.4, переход к версии OpenGL 2.0 означался появлением новой важной особенностью – поддержкой программных шейдеров. Эта поддержка была перенесена в OpenGL ES 2.0 и стала одним из основных элементов спецификации WebGL 1.0.

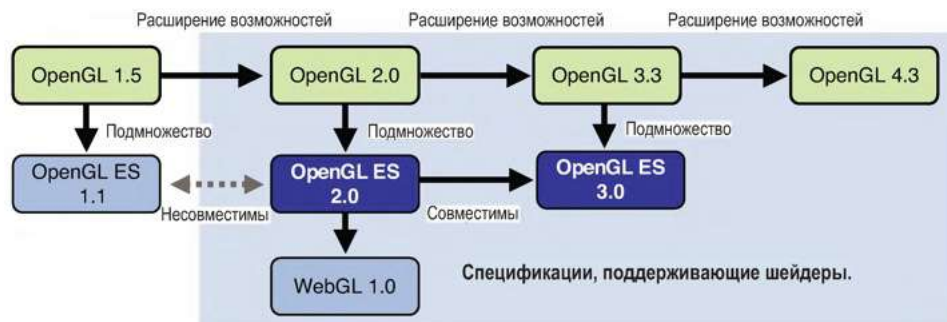


Рис. 1.4. Связь между OpenGL, OpenGL ES 1.1/2.0/3.0 и WebGL

Программные шейдеры, или просто шейдеры, – это компьютерные программы, позволяющие создавать сложные визуальные эффекты с использованием специализированного языка программирования, похожего на язык C. В этой книге мы подробно будем рассказывать о программировании шейдеров, что позволит вам быстро овладеть технологией WebGL. Язык программирования, используемый для создания шейдеров, называется языком шейдеров (shading language). Язык шейдеров, определяемый в спецификации OpenGL ES 2.0, основан на языке шейдеров OpenGL (GLSL) и называется языком шейдеров OpenGL ES (GLSL ES). Так как WebGL основана на OpenGL ES 2.0, в ней, для создания шейдеров, так же используется язык GLSL ES.

За развитие и стандартизацию OpenGL отвечает Khronos Group (некоммерческий промышленный консорциум, образованный для разработки, публикации и продвижения различных открытых стандартов). В 2009 году консорциум Khronos учредил рабочую группу WebGL и запустил процесс стандартизации WebGL на основе OpenGL ES 2.0. В 2011 под его эгидой была выпущена первая версия WebGL. Эта книга написана в первую очередь на основе этой спецификации и, там где это необходимо, упоминается последняя версия спецификации WebGL, опубликованной как рабочая версия. За дополнительной информацией обращайтесь к тексту спецификации¹.

Структура приложений на основе WebGL

Динамические веб-страницы обычно создаются как комбинации HTML и JavaScript. С появлением WebGL, появилась необходимость добавить в эту комбинацию язык шейдеров GLSL ES. Это означает, что веб-страницы, использующие технологию WebGL, создаются на трех языках: HTML5 (гипертекстовый язык разметки), JavaScript и GLSL ES. На рис. 1.5 представлена структура традиционной динамической веб-страницы (слева) и веб-страницы, использующей WebGL (справа).

¹ WebGL 1.0: www.khronos.org/registry/webgl/specs/1.0/; рабочая версия: www.khronos.org/registry/webgl/specs/latest/

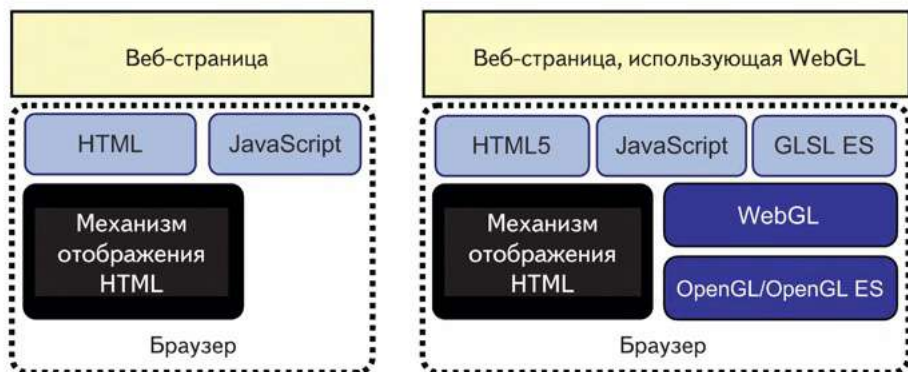
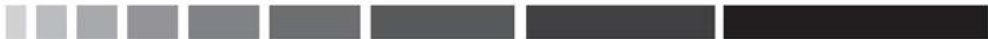


Рис. 1.5. Структура типичной динамической веб-страницы (слева) и веб-страницы, использующей WebGL (справа)

Однако, поскольку GLSL ES вообще написан на JavaScript, для разработки приложений на основе WebGL фактически используются только два языка: HTML и JavaScript. То есть, несмотря на то, что WebGL увеличивает сложность сценариев на JavaScript, структура приложений остается той же, что и структура динамических веб-страниц, – используются только файлы HTML и JavaScript.

В заключение

В этой главе мы дали краткий обзор WebGL, рассказали о некоторых ключевых особенностях и схематически обозначили структуру приложений WebGL. Главный вывод, следующий из этой главы: приложения на основе WebGL разрабатываются с использованием трех языков – HTML5, JavaScript и GLSL ES, – однако, из-за того, что язык шейдеров (GLSL ES) вообще встроен в JavaScript, приложения WebGL имеют в точности ту же структуру, что и традиционные веб-страницы. В следующей главе мы расскажем, как создавать приложения с использованием WebGL, и подробно опишем ряд простых примеров.



ГЛАВА 2.

Первые шаги в WebGL

Как рассказывалось в главе 1 «Обзор WebGL», для создания WebGL-приложений, рисующих трехмерную графику на экране, используют два языка: HTML и JavaScript. Сама система WebGL использует для рисования новый элемент `<canvas>`, появившийся в HTML5 и определяющий область веб-страницы, предназначенную для рисования. Без использования технологии WebGL, элемент `<canvas>` позволяет рисовать только 2-мерную графику, с применением функций JavaScript. Используя WebGL, в том же самом элементе можно рисовать уже трехмерную графику.

В этой главе мы поближе познакомимся с элементом `<canvas>` и основными функциями WebGL, детально исследовав процесс создания нескольких примеров программ. Все примеры написаны на JavaScript и используют WebGL для отображения простой фигуры и организации взаимодействий с ней. По этой причине такие программы на JavaScript обычно называют **WebGL-приложениями**.

Примеры WebGL-приложений подчеркивают некоторые ключевые моменты, включая:

- порядок использования элемента `<canvas>` в WebGL-приложениях и рисования в нем;
- связь между HTML и WebGL посредством JavaScript;
- простые функции рисования WebGL;
- роль шейдеров в WebGL.

К концу этой главы у вас должно сложиться понимание, как писать и выполнять простые WebGL-приложения, и как рисовать простые 2-мерные фигуры. Эти знания вам пригодятся в следующих главах, где будет продолжено исследование основ WebGL: в главе 3 «Рисование и преобразование треугольников», в главе 4, «Дополнительные преобразования и простая анимация» и в главе 5 «Цвет и текстура».

Что такое canvas?

До появления HTML5, если требовалось вывести изображение в веб-странице, единственным стандартным способом сделать это было использование тега `<img>`. Этот тег, хотя и является довольно удобным инструментом, имеет множество

ограничений и не позволяет создавать и выводить изображения динамически. Это стало одной из основных причин появления сторонних решений, таких как Flash Player.

Однако тег `<canvas>` в HTML5 изменил положение дел, дав удобную возможность рисовать компьютерную графику динамически с помощью JavaScript.

Подобно холсту (canvas) художника, тег `<canvas>` определяет область рисования в веб-странице. Только вместо кистей и красок, для рисования в этой области используется JavaScript. Вы можете рисовать точки, линии, прямоугольники, окружности и другие геометрические фигуры, вызывая методы JavaScript, поддерживаемые для тега `<canvas>`. На рис. 2.1 показан простенький графический редактор, использующий тег `<canvas>`.

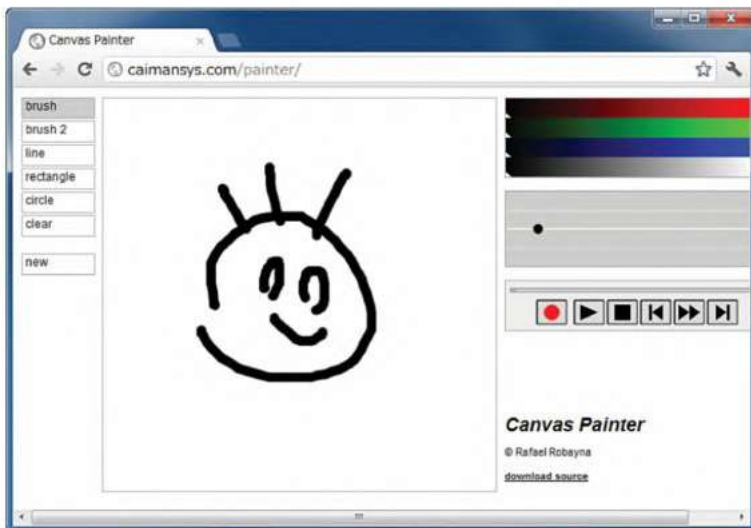


Рис. 2.1. Простой графический редактор, использующий тег `<canvas>` (<http://caimansys.com/painter/>)

Этот графический редактор представляет собой веб-страницу и позволяет в интерактивном режиме рисовать линии, прямоугольники и окружности, и даже изменять их цвет.

Мы пока не будем создавать что-то сложное, а просто рассмотрим основные функции тега `<canvas>`, воспользовавшись примером программы `DrawRectangle`, которая рисует закрашенный синий квадрат на веб-странице. На рис. 2.2 показан результат выполнения программы `DrawRectangle` в браузере.

Использование тега `<canvas>`

Итак, посмотрим, как действует программа `DrawRectangle` и выясним, как использовать тег `<canvas>` в файле HTML. В листинге 2.1 приводится содержимое файла `DrawingTriangle.html`. Обратите внимание, что все файлы HTML в этой книге написаны на HTML5.

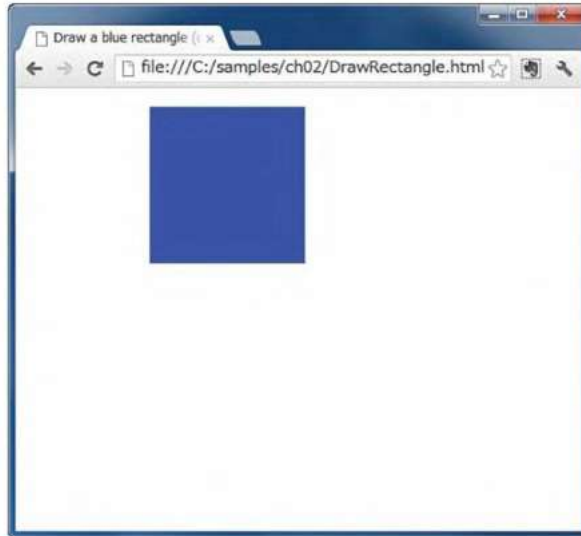


Рис. 2.2. Программа DrawRectangle

Листинг 2.1. DrawRectangle.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8" />
5     <title>Draw a blue rectangle (canvas version)</title>
6   </head>
7
8   <body onload="main()">
9     <canvas id="example" width="400" height="400">
10       Please use a browser that supports "canvas"
11     </canvas>
12     <script src="DrawRectangle.js"></script>
13   </body>
14 </html>
```

Тег `<canvas>` находится в строке 9. С помощью своих атрибутов `width` и `height` он определяет область рисования размером 400×400 пикселей. Атрибут `id` определяет идентификатор этой области и будет использоваться далее:

```
<canvas id="example" width="400" height="400"></canvas>
```

По умолчанию область рисования невидима (фактически, она прозрачна), пока в ней не будет что-нибудь нарисовано, что мы и собираемся сделать с помощью JavaScript. Это все, что необходимо сделать в файле HTML, чтобы подготовить `<canvas>` для использования WebGL-программой. Следует, однако, заметить, что данная строка будет иметь эффект, только в браузерах, поддерживающих тег `<canvas>` – браузеры, не имеющие такой поддержки, будут просто игнорировать

эту строку и на экране ничего не появится. Чтобы как-то решить эту проблему, можно вывести сообщение об ошибке, как показано ниже:

```

9      <canvas id="example" width="400" height="400">
10     Please use a browser that supports "canvas"
11     </canvas>

```

Чтобы что-то нарисовать в области, определяемой тегом `<canvas>`, необходим код на JavaScript, выполняющий операции рисования. Соответствующий код можно включить непосредственно в файл HTML или сохранить его в отдельном файле. В наших примерах мы будем использовать второй подход, потому что так проще будет читать программный код. Независимо от выбранного подхода, нужно сообщить браузеру, как запустить код на JavaScript. Именно это мы и делаем в строке 8, сообщая браузеру, что после загрузки файла с кодом на JavaScript он должен вызвать функцию `main()`. Реализовано это с помощью атрибута `onload` элемента `<body>`, который предписывает браузеру вызвать функцию `main()` после загрузки элемента `<body>`:

```
8  <body onload="main()">
```

Строка 12 определяет, что браузер должен импортировать файл `DrawRectangle.js` с кодом на JavaScript, где определена функция `main()`:

```
12  <script src="DrawRectangle.js"></script>
```

Для простоты, во всех примерах программ, что приводятся в этой книге, файлам JavaScript присвоены те же имена, что и соответствующим им файлам HTML (см. рис. 2.3).

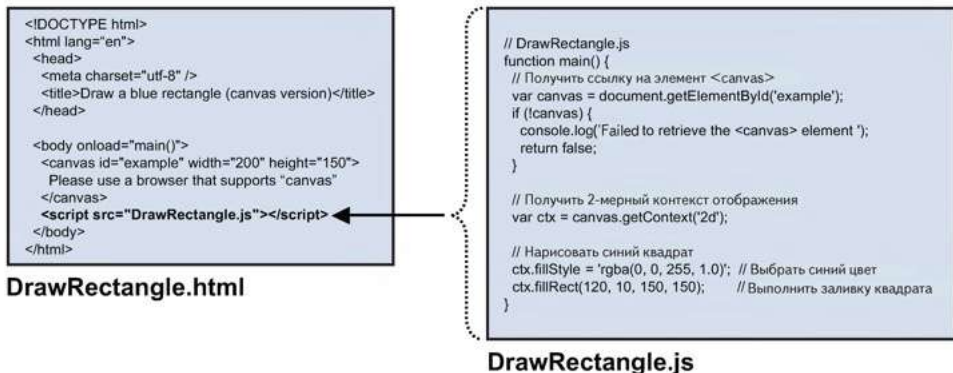


Рис. 2.3. DrawRectangle.html и DrawRectangle.js

DrawRectangle.js

Файл `DrawRectangle.js` хранит код программы на языке JavaScript, рисующей синий квадрат в области рисования, определяемой тегом `<canvas>` (см.

листинг 2.2). Он содержит всего 16 строк, выполняющих три этапа рисования двумерной графики:

1. Получить ссылку на элемент `<canvas>`.
2. Запросить контекст отображения двумерной графики.
3. Нарисовать двумерное изображение, используя методы контекста.

Эти три этапа необходимо выполнить при рисовании любой графики, двух- или трехмерной. В данном случае выполняется рисование простого двумерного квадрата с применением стандартных функций JavaScript. При рисовании трехмерной графики с использованием возможностей WebGL на втором этапе (строка 11) мы запросили бы трехмерный контекст вместо двумерного; однако сама последовательность этапов осталась бы той же самой.

Листинг 2.2. DrawRectangle.js

```
1 // DrawRectangle.js
2 function main() {
3   // Получить ссылку на элемент <canvas>                                <- (1)
4   var canvas = document.getElementById('example');
5   if (!canvas) {
6     console.log('Failed to retrieve the <canvas> element');
7     return;
8   }
9
10  // Получить 2-мерный контекст отображения                             <- (2)
11  var ctx = canvas.getContext('2d');
12
13  // Нарисовать синий квадрат                                           <- (3)
14  ctx.fillStyle = 'rgba(0, 0, 255, 1.0)'; // Выбрать синий цвет
15  ctx.fillRect(120, 10, 150, 150);    // Выполнить заливку квадрата
16 }
```

Рассмотрим эти этапы по порядку.

Получить ссылку на элемент `<canvas>`

Чтобы что-то нарисовать в области рисования, определяемой элементом `<canvas>`, необходимо сначала получить ссылку на него в программе JavaScript. Сделать это можно с помощью метода `document.getElementById()`, как показано в строке 4. Этот метод имеет единственный параметр – строковый идентификатор, определяемый атрибутом `id` в теге `<canvas>`. В данном случае используется идентификатор `'example'`, как определено в файле `DrawRectangle.html`, в строке 9 (см. листинг 2.1).

Если метод вернет значение, отличное от `null`, значит вам удалось получить ссылку на искомый элемент. В противном случае попытка считается неудачной. Проверить успешность попытки можно с помощью простой инструкции `if`, как показано в строке 5. В данном примере, в случае неудачи выполняется строка 6. В ней вызывается метод `console.log()`, который выводит свой параметр в виде строки в консоль браузера.

Примечание. В браузере Chrome консоль можно открыть, выбрав пункт меню **Tools** → **JavaScript Console** (Инструменты → Консоль JavaScript) или нажав комбинацию клавиш **Ctrl+Shift+J** (см. рис. 2.4); в Firefox то же самое можно сделать, выбрав пункт меню **Tools** → **Web Developer** → **Web Console** (Инструменты → Веб-разработка → Веб-консоль) или нажав комбинацию клавиш **Ctrl+Shift+K**.

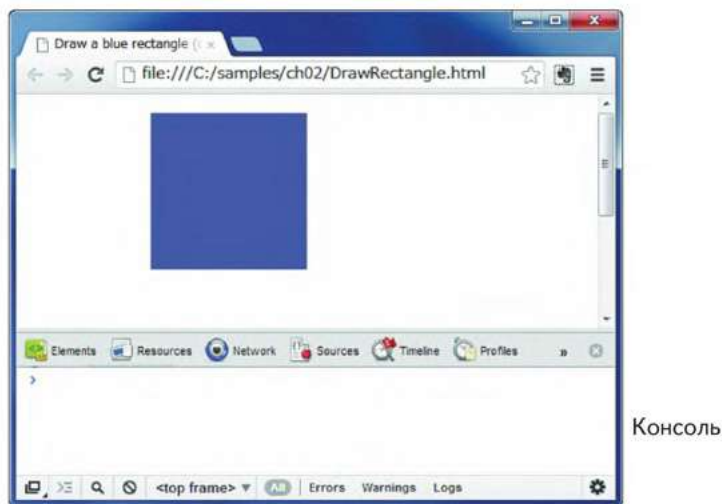


Рис. 2.4. Консоль в Chrome

Запросить контекст отображения двумерной графики

Так как элемент `<canvas>` изначально предусматривает возможность отображения двух- и трехмерной графики, он не предоставляет методы рисования непосредственно, а дает доступ к механизму, который называют *контекстом*, поддерживающим фактические средства рисования. Мы получаем этот контекст в строке 11:

```
11 var ctx = canvas.getContext('2d');
```

Метод `canvas.getContext()` имеет параметр, определяющий тип требуемого механизма рисования. Данный пример рисует двумерную фигуру, поэтому методу передается строка `'2d'` (будьте внимательны, регистр символов имеет значение).

В результате этого вызова мы получаем контекст и сохраняем его в переменной `ctx`. Отметьте, что для экономии места мы не выполняем проверку ошибок, которую обязательно следует делать в действующих программах.

Нарисовать двумерное изображение, используя методы контекста

Теперь, получив контекст отображения, давайте посмотрим, как выполняется рисование синего квадрата. Делается это в два этапа. Сначала устанавливается цвет, а затем этим цветом рисуется (или заливается) сам квадрат.

Эти два этапа выполняют строки 14 и 15:

```
13 // Нарисовать синий квадрат
14 ctx.fillStyle = 'rgba(0, 0, 255, 1.0)'; // Выбрать синий цвет
15 ctx.fillRect(120, 10, 150, 150);      // Выполнить заливку квадрата
```

Ключевое слово `rgba` в строковом значении `'rgba(0, 0, 255, 1.0)'` в строке 14 определяет четыре составляющие цвета: `r` (red – красный), `g` (green – зеленый), `b` (blue – синий) и `alpha` – альфа-канал: `transparency` – прозрачность), каждая из которых может иметь значение от 0 (наименьшее) до 255 (наибольшее), кроме альфа-канала, значение которого определяется числом в диапазоне от 0.0 (прозрачный) до 1.0 (непрозрачный). Вообще цвет в компьютерных системах обычно представлен комбинацией красной, зеленой и синей составляющих (три основных цвета), – этот формат называют **RGB**. При добавлении альфа-составляющей (прозрачность), формат называют **RGBA**.

В строке 14 используется свойство `fillStyle`, определяющее цвет заливки. Однако, прежде чем погрузиться в детали аргументов в строке 15, рассмотрим систему координат, используемую элементом `<canvas>` (см. рис. 2.5).

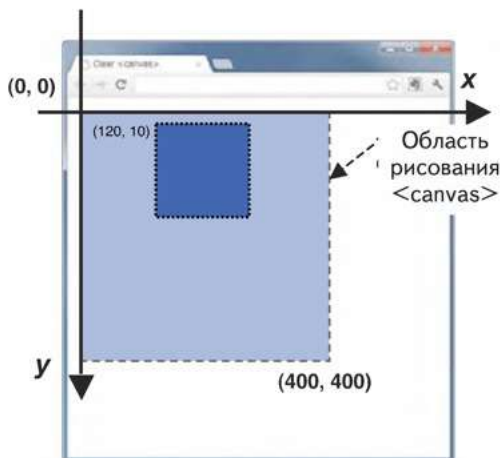


Рис. 2.5. Система координат элемента `<canvas>`

Как показано на рис. 2.5, начало системы координат элемента `<canvas>` находится в верхнем левом углу, ось `X` определяет координату по горизонтали (слева направо), а ось `Y` – координату по вертикали (сверху вниз). Пунктирная линия на рис. 2.5 показывает границы области рисования определяемой элементом `<canvas>` в файле HTML (см. листинг 2.1), которая имеет размер 400 на 400 пикселей. Линия из точек – это границы квадрата, который рисует программа.

Первые два параметра метода `ctx.fillRect()` определяют координаты левого верхнего угла рисуемого квадрата в системе координат элемента `<canvas>`, а третий и четвертый параметры – ширину и высоту квадрата (в пикселях):

```
15 ctx.fillRect(120, 10, 150, 150);      // Выполнить заливку квадрата
```

После того, как браузер загрузит файл `DrawRectangle.html`, вы увидите квадрат, как показано на рис. 2.2.

Итак, мы увидели, как рисовать двухмерную графику. Однако, WebGL использует тот же самый элемент `<canvas>` для рисования трехмерной графики, поэтому давайте войдем в мир WebGL.

Самая короткая WebGL-программа: очистка области рисования

Знакомство с миром WebGL мы начнем с самой короткой программы, `HelloCanvas`, которая просто очищает область рисования, определяемую тегом `<canvas>`. На рис. 2.6 показан результат загрузки программы, очищающей прямоугольную область простой заливкой ее черным цветом.

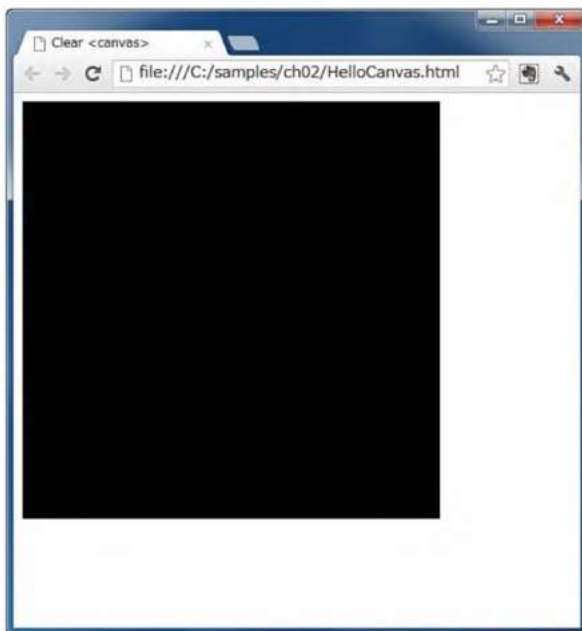


Рис. 2.6. `HelloCanvas`

Файл HTML (*`HelloCanvas.html`*)

Рассмотрим файл `HelloCanvas.html`, представленный на рис. 2.7. Он имеет простую структуру и начинается с определения области рисования в виде элемента `<canvas>`, в строке 9, и затем импортирует `HelloCanvas.js` (WebGL-программу) в строке 16.

В строках с 13 по 15 импортируется еще несколько файлов JavaScript, в которых определяются удобные вспомогательные функции, упрощающие программирова-

ние графики WebGL. Подробнее эти файлы будут описываться позже, а пока просто считайте их библиотеками.

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="utf-8" />
5   <title>Clear canvas</title>
6 </head>
7
8 <body onload="main()">
9   <canvas id="webgl" width="400" height="400">
10     Please use the browser supporting "canvas"
11   </canvas>
12
13   <script src="../lib/webgl-utils.js"></script>
14   <script src="../lib/webgl-debug.js"></script>
15   <script src="../lib/cuon-utils.js"></script>
16   <script src="HelloCanvas.js"></script>
17 </body>
18 </html>
```

Элемент `<canvas>`, в котором механизм WebGL рисует фигуры

Файлы JavaScript со вспомогательными функциями для работы с WebGL

Файл JavaScript, реализующий рисование фигур в элементе `<canvas>`

Рис. 2.7. HelloCanvas.html

В строке 9 выполняется настройка элемента `<canvas>` и затем, в строке 16, импортируется файл `HelloCanvas.js`, который фактически использует команды WebGL для доступа к области рисования. Рассмотрим теперь WebGL-программу в файле `HelloCanvas.js`.

Программа на JavaScript (HelloCanvas.js)

Файл `HelloCanvas.js` (см. листинг 2.3) содержит всего 18 строк кода, включая комментарии и операции по обработке ошибок, и выполняет те же 3 этапа, которые описывались, когда мы рассматривали пример рисования двухмерной графики: получает ссылку на элемент `<canvas>`, получает контекст отображения и осуществляет рисование.

Листинг 2.3. HelloCanvas.js

```
1 // HelloCanvas.js
2 function main() {
3   // Получить ссылку на элемент <canvas>
4   var canvas = document.getElementById('webgl');
5
6   // Получить контекст отображения для WebGL
7   var gl = getWebGLContext(canvas);
```



```
8  if (!gl) {  
9      console.log('Failed to get the rendering context for WebGL');  
10     return;  
11 }  
12  
13 // Указать цвет для очистки области рисования <canvas>  
14 gl.clearColor(0.0, 0.0, 0.0, 1.0);  
15  
16 // Очистить <canvas>  
17 gl.clear(gl.COLOR_BUFFER_BIT);  
18 }
```

Как и в предыдущем примере, здесь определена единственная функция `main()`, на которую ссылается HTML-файл `HelloCanvas.html` в атрибуте `onload` элемента `<body>` (строка 8 на см. рис. 2.7).

На рис. 2.8 показано, как действует функция `main()` в нашей WebGL-программе, выполняющая четыре шага, которые обсуждаются далее по очереди.



Рис. 2.8. Порядок работы функции `main()`

Получить ссылку на элемент `<canvas>`

Сначала функция `main()` получает ссылку на элемент `<canvas>` в файле HTML. Как описывалось в обсуждении программы `DrawRectangle.js`, для этого вызывается метод `document.getElementById()`, которому в качестве аргумента передается идентификатор `'webgl'` элемента. Если взглянуть на содержимое файла `HelloCanvas.html` (см. рис. 2.7), можно увидеть, что этот идентификатор определен в атрибуте `id` тега `<canvas>`, в строке 9:

```
9  <canvas id='webgl' width='400' height='400'>
```

Значение, возвращаемое этим методом, сохраняется в переменной `canvas`.

Получить контекст отображения для WebGL

На следующем шаге программа использует переменную `canvas` с целью получения контекста отображения для WebGL. Для получения контекста WebGL, можно было бы использовать метод `canvas.getContext()`, как описывалось выше. Однако, из-за того, что в разных браузерах аргумент, который требуется передать методу `canvas.getContext()`, может отличаться¹, мы решили написать специальную функцию `getWebGLContext()`, чтобы скрыть различия между браузерами:

```
7 var gl = getWebGLContext(canvas);
```

Это – одна из вспомогательных функций, упоминавшихся выше и специально написанных для данной книги, в файле `cuon-utils.js`, импортируемом в строке 15, в файле `HelloCanvas.html`. Функции становятся доступными после загрузки файла, путь к которому определяется атрибутом `src` тега `<script>`. Ниже приводится описание функции `getWebGLContext()`.

`getWebGLContext(element [, debug])`

Возвращает контекст отображения для WebGL, устанавливает режим отладки и выводит любые сообщения об ошибках в консоль браузера.

Параметры	<code>element</code>	Определяет запрашиваемый элемент <code><canvas></code> .
	<code>debug</code> (необязательный)	Значение по умолчанию <code>true</code> . Когда установлен в значение <code>true</code> , ошибки JavaScript выводятся в консоль браузера. Примечание: выключайте после отладки, так как оказывает отрицательное влияние на производительность.
Возвращаемое значение	непустое значение	Контекст отображения для WebGL.
	<code>null</code>	WebGL не поддерживается браузером.

Порядок получения ссылки на элемент `<canvas>` и использование ее для получения контекста отображения остается тем же, что и в программе `DrawRectangle.js`, показанной выше, где контекст отображения использовался для рисования двухмерной графики.

Точно так же для рисования в области элемента `<canvas>` WebGL использует контекст отображения, возвращаемый функцией `getWebGLContext()`. Однако теперь контекст поддерживает возможность рисования трехмерной графики вместо двухмерной, то есть предоставляет доступ к методам WebGL. В строке 7 программа сохраняет контекст в переменной `gl`. Вы можете использовать в своих программах любые другие имена переменных. Мы преднамеренно выбрали имя `gl` для использования в примерах к этой книге, потому что это делает вызовы методов WebGL больше похожими на вызовы методов, определяемых спецификацией

¹ В большинстве браузеров можно передать аргумент `'experimental-webgl'`. Но со временем предполагается заменить это значение на `'webgl'`, поэтому мы решили скрыть этот факт за интерфейсом функции.

OpenGL ES 2.0, которая является основой для спецификации WebGL. Например, сравните вызов `gl.clearColor()` в строке 14 с именем метода `glClearColor()` из OpenGL ES 2.0 или OpenGL:

```
14 gl.clearColor(0.0, 0.0, 0.0, 1.0);
```

В этой книге описываются все методы WebGL, и повсюду будет предполагаться, что переменная, хранящая ссылку на контекст, имеет имя `gl`.

После получения контекста отображения для WebGL, следующим шагом является установка цвета с помощью контекста для очистки области рисования, определяемой элементом `<canvas>`.

Указать цвет для очистки области рисования `<canvas>`

В предыдущем разделе, программа `DrawRectangle.js` устанавливала цвет перед рисованием квадрата. Аналогично WebGL-программа должна установить цвет перед очисткой области рисования. В строке 14 вызывается метод `gl.clearColor()`, устанавливающий цвет в формате RGBA.

`gl.clearColor(red, green, blue, alpha)`

Устанавливает цвет для очистки (заливки) области рисования.

Параметры	<code>red</code>	Определяет значение красной составляющей цвета (от 0.0 до 1.0).
	<code>green</code>	Определяет значение зеленой составляющей цвета (от 0.0 до 1.0).
	<code>blue</code>	Определяет значение синей составляющей цвета (от 0.0 до 1.0).
	<code>alpha</code>	Определяет значение альфа-канала (прозрачность) цвета (от 0.0 до 1.0). Значение 0.0 соответствует полностью прозрачному цвету, значение 1.0 – полностью непрозрачному.
Любые значения меньше 0.0 и больше 1.0 в этих параметрах усекаются до 0.0 и 1.0, соответственно.		

Возвращаемое значение нет

Ошибки* нет

* В этой книге в описаниях всех методов WebGL будет присутствовать пункт «Ошибки», в котором мы будем отмечать ошибки, которые не могут быть определены по возвращаемому значению. По умолчанию ошибки не выводятся, но могут выводиться в консоль JavaScript, установкой второго аргумента метода `getWebGLContext()` в значение `true`.

Возможно, вы заметили, что в нашем примере рисования двумерной фигуры, `DrawRectangle`, значения составляющих цвета определялись числами в диапазоне от 0 до 255. Однако, так как технология WebGL основана на OpenGL, в ней используются значения в диапазоне от 0.0 до 1.0, традиционные для OpenGL. Чем больше значение, тем выше интенсивность составляющей цвета. Аналогично, бо-

лее высокое значение параметра `alpha` (четвертый параметр) соответствует меньшей прозрачности (то есть, большей непрозрачности).

После того, как программа укажет цвет для очистки, он сохраняется в системе WebGL и не изменяется, пока не будет указан другой цвет вызовом `gl.clearColor()`. Это означает, что не требуется определять цвет для очистки, если в какой-то момент потребуется очистить область рисования.

Очистить <canvas>

Наконец, можно вызвать метод `gl.clear()`, чтобы очистить область рисования выбранным цветом:

```
17 gl.clear(gl.COLOR_BUFFER_BIT);
```

Обратите внимание, что аргументом этого метода является значение `gl.COLOR_BUFFER_BIT`, а не ссылка на элемент `<canvas>`, как можно было бы ожидать. Это объясняется тем, что метод `gl.clear()` фактически опирается на спецификацию OpenGL, которая использует более сложную модель, чем простые области рисования, манипулируя множеством внутренних буферов. Один такой буфер – буфер цвета – используется в данном примере. Указав значение `gl.COLOR_BUFFER_BIT`, мы сообщили системе WebGL, что для очистки области рисования он должен использовать буфер цвета. Помимо буфера цвета в WebGL имеется еще множество буферов, включая буфер глубины и буфер трафарета. Подробнее о буфере цвета рассказывается далее в этой главе, а о буфере глубины – в главе 7, «Вперед, в трехмерный мир». Буфер трафарета в этой книге рассматриваться не будет, потому что он редко используется на практике.

Очистка буфера цвета фактически вынуждает WebGL очистить область рисования `<canvas>` на веб-странице.

`gl.clear(buffer)`

Очищает указанный буфер предварительно определенными значениями. В случае буфера цвета, значение (цвет) устанавливается вызовом метода `gl.clearColor()`.

Параметры	<code>buffer</code>	Определяет очищаемый буфер. С помощью оператора <code> </code> (поражающая операция «ИЛИ») можно указать несколько буферов.
	<code>gl.COLOR_BUFFER_BIT</code>	Определяет буфер цвета.
	<code>gl.DEPTH_BUFFER_BIT</code>	Определяет буфер глубины.
	<code>gl.STENCIL_BUFFER_BIT</code>	Определяет буфер трафарета.

Возвращаемое значение нет

Ошибки `INVALID_VALUE` Аргумент `buffer` имеет значение, отличное от любого из трех, указанных выше.

Если цвет не был указан, то есть, если метод `gl.clearColor()` еще не вызывался, используются значения по умолчанию, перечисленные в табл. 2.1.

Таблица 2.1. Значения по умолчанию для очистки буферов и соответствующие методы установки значений очистки

Буфер	Значение по умолчанию	Метод установки
Буфер цвета	(0.0, 0.0, 0.0, 0.0)	<code>gl.clearColor(red, green, blue, alpha)</code>
Буфер глубины	1.0	<code>gl.clearDepth(depth)</code>
Буфер трафарета	0	<code>gl.clearStencil(s)</code>

Теперь, когда мы подробно разобрали этот простой пример WebGL-программы, загрузите `HelloCanvas` в свой браузер, чтобы на практике убедиться, что область рисования действительно окрашивается в черный цвет. Помните, что все примеры программ в книге можно запускать непосредственно со вспомогательного веб-сайта. Однако, если вы пожелаете поэкспериментировать с какой-нибудь из программ, вам потребуется загрузить архив с примерами на свой локальный диск. Если вы уже сделали это, чтобы загрузить пример нужно перейти в каталог с примером на диске и открыть файл `HelloCanvas.html` в браузере.

Эксперименты с примером программы

Поэкспериментируйте с примером программы, чтобы лучше усвоить, как определяются цвета в WebGL. Для этого попробуйте изменить цвет очистки. С помощью простого текстового редактора измените строку 14 в файле `HelloCanvas.js`, как показано ниже и сохраните изменения:

```
14 gl.clearColor(0.0, 0.0, 1.0, 1.0);
```

После перезагрузки `HelloCanvas.html` в браузер, файл `HelloCanvas.js` также перезагрузится, затем будет вызвана функция `main()`, которая очистит область рисования, окрасив ее в синий цвет. Попробуйте указать другие цвета и посмотрите, что из этого получается. Например, после вызова `gl.clearColor(0.5, 0.5, 0.5, 1.0)` область будет залита серым цветом.

Рисование точки (версия 1)

В предыдущем разделе мы видели, как инициализировать контекст WebGL и как использовать некоторые его методы. В этом разделе мы сделаем еще один шаг вперед и напишем программу рисования самой простой фигуры: точки. Программа будет рисовать красную точку размером 10 пикселей в позиции с координатами (0.0, 0.0, 0.0). Так как система WebGL предназначена для работы с трехмерной графикой, позиция точки определяется тремя координатами. Описание системы координат будет представлено позже, а пока просто имейте в виду, что точка с координатами (0.0, 0.0, 0.0) соответствует центру области рисования `<canvas>`.

Пример программы называется `HelloPoint1` и, как показано на рис. 2.9, эта программа рисует красную точку (прямоугольник) в центре области рисования

<canvas>, которая очищается черным цветом². В качестве точек вы часто будете использовать закрашенные прямоугольники вместо кругов, потому что рисование прямоугольника выполняется быстрее. (Как рисовать круглые точки мы расскажем в главе 9, «Иерархические объекты».)

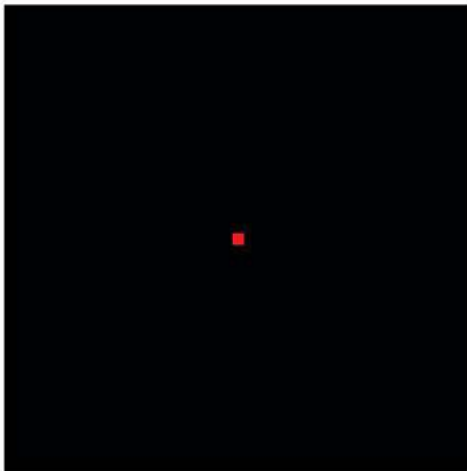


Рис. 2.9. HelloPoint1

Как в случае с очисткой области рисования в предыдущем разделе, цвет точки требуется указать в формате RGBA. Чтобы получить красный цвет, составляющая R должна иметь значение 1.0, составляющая G – значение 0.0, B – 0.0 и A – 1.0. Как вы наверняка помните, в примере `DrawRectangle.js`, приводившемся выше в этой главе, сначала выполняется установка цвета, а затем осуществляется рисование прямоугольника:

```
ctx.fillStyle='rgba(0, 0, 255, 1.0)';  
ctx.fillRect(120, 10, 150, 150);
```

Соответственно, кто-то из вас мог бы предположить, что WebGL-приложению подобные операции выполняются некоторым похожим образом, например:

```
gl.drawColor(1.0, 0.0, 0.0, 1.0);  
gl.drawPoint(0, 0, 0, 10);           // Координаты центра и размеры точки
```

К сожалению это не так. WebGL опирается на механизм рисования нового типа, который называется **шейдером** (shader), обладающий большей гибкостью и ши-

² Примеры программ в главе 2 написаны максимально простыми, чтобы читатель мог сосредоточиться на освоении функциональности шейдеров. В частности, в примерах не используются «буферные объекты» (см. главу 3), которые широко применяются в WebGL. Хотя это и помогает упростить обучение, некоторые браузеры (в частности Firefox) могут неправильно отображать примеры без использования буферных объектов. В последующих главах и в действующих приложениях такая проблема не возникает, потому что в этом случае «буферные объекты» используются. Поэтому, если у вас будут возникать проблемы с примерами из этой главы, попробуйте открывать их в другом браузере. Вернуться к своему любимому браузеру вы сможете уже в следующей главе.

ротой возможностей при рисовании двух- и трехмерных объектов, и который должен использоваться всеми WebGL-приложениями. Однако шейдеры – не только мощный, но также очень сложный механизм, и при его применении невозможно просто использовать элементарные операции рисования.

Так как шейдеры являются критически важным механизмом в программировании с применением технологии WebGL, мы будем двигаться дальше небольшими шагами, чтобы вы могли лучше понять их.

>HelloPoint1.html

В листинге 2.4 приводится содержимое файла HelloPoint1.html, являющегося функциональным эквивалентом файла HelloCanvas.html (см. листинг 2.7). Заголовок веб-страницы и имя файла с программным кодом на JavaScript здесь изменились (строки 5 и 16), но все остальное осталось прежним. С этого момента мы будем опускать листинги с файлами HTML, если только они не отличаются от данного примера.

Листинг 2.4. HelloPoint1.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8" />
5     <title>Draw a point (1)</title>
6   </head>
7
8   <body onload="main()">
9     <canvas id="webgl" width="400" height="400">
10      Please use the browser supporting "canvas".
11    </canvas>
12
13    <script src="../libs/webgl-utils.js"></script>
14    <script src="../libs/webgl-debug.js"></script>
15    <script src="../libs/cuon-utils.js"></script>
16    <script src="HelloPoint1.js"></script>
17  </body>
18 </html>
```

>HelloPoint1.js

В листинге 2.5 приводится содержимое файла HelloPoint1.js. Как следует из комментариев, в нем определены два «шейдера» (строки со 2 по 13). Посмотрите, как определены шейдеры, и переходите к следующему разделу, где приводится подробное их описание.

Листинг 2.5. HelloPoint1.js

```
1 // HelloPoint1.js
2 // Вершинный шейдер
```

```

3 var VSHADER_SOURCE =
4   'void main() {\n' +
5   '   gl_Position = vec4(0.0, 0.0, 0.0, 1.0);\n' + // Координаты
6   '   gl_PointSize = 10.0;\n' +                // Установить размер точки
7   '}\n';
8
9 // Фрагментный шейдер
10 var FSHADER_SOURCE =
11   'void main() {\n' +
12   '   gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);\n' + // Установить цвет
13   '}\n';
14
15 function main() {
16   // Получить ссылку на элемент <canvas>
17   var canvas = document.getElementById('webgl');
18
19   // Получить контекст отображения для WebGL
20   var gl = getWebGLContext(canvas);
21   if (!gl) {
22     console.log('Failed to get the rendering context for WebGL');
23     return;
24   }
25
26   // Инициализировать шейдеры
27   if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
28     console.log('Failed to initialize shaders.');
```

```

29     return;
30   }
31
32   // Указать цвет для очистки <canvas>
33   gl.clearColor(0.0, 0.0, 0.0, 1.0);
34
35   // Очистить <canvas>
36   gl.clear(gl.COLOR_BUFFER_BIT);
37
38   // Нарисовать точку
39   gl.drawArrays(gl.POINTS, 0, 1);
40 }
```

Что такое шейдер?

HelloPoint1.js – это наша первая WebGL-программа, использующая шейдеры. Как уже говорилось выше, шейдеры необходимы, когда требуется что-то нарисовать с помощью WebGL. По сути, шейдер – это программа, «встроенная» в программу на языке JavaScript, и в данном случае шейдеры определяются в самом начале программы. На первый взгляд они выглядят сложными, но давайте разберемся не спеша, так ли это.

Для рисования с помощью WebGL требуется два типа шейдеров, определения которых можно видеть в строках 2 и 9:

- **вершинный шейдер (vertex shader):** вершинный шейдер – это программа, описывающая характеристики вершины (координаты, цвет и другие), а

вершина – это точка в двух- или трехмерном пространстве, например, угол или вершина двух- или трехмерной фигуры;

- **фрагментный шейдер (fragment shader):** фрагментный шейдер – это программа, реализующая обработку фрагментов изображений, например, определение освещенности (см. главу 8, «Освещение объектов»), где под фрагментом подразумевается простейший элемент изображения, своего рода «пиксель».

Позднее мы подробно обсудим шейдеры. Одной лишь трехмерной сцены недостаточно для рисования графики. Необходимо так же учитывать освещенность объектов и изменение позиции зрителя. Шейдеры обеспечивают в этом отношении большую гибкость и высокую реалистичность современной трехмерной графики, позволяя использовать новые эффекты отображения для достижения ошеломляющих результатов.

Шейдеры извлекаются из кода на JavaScript и сохраняются в системе WebGL, готовыми к использованию. На рис. 2.10 показана типовая последовательность операций – от запуска программы JavaScript до передачи шейдеров системе WebGL, – которая применяется при рисовании фигур.

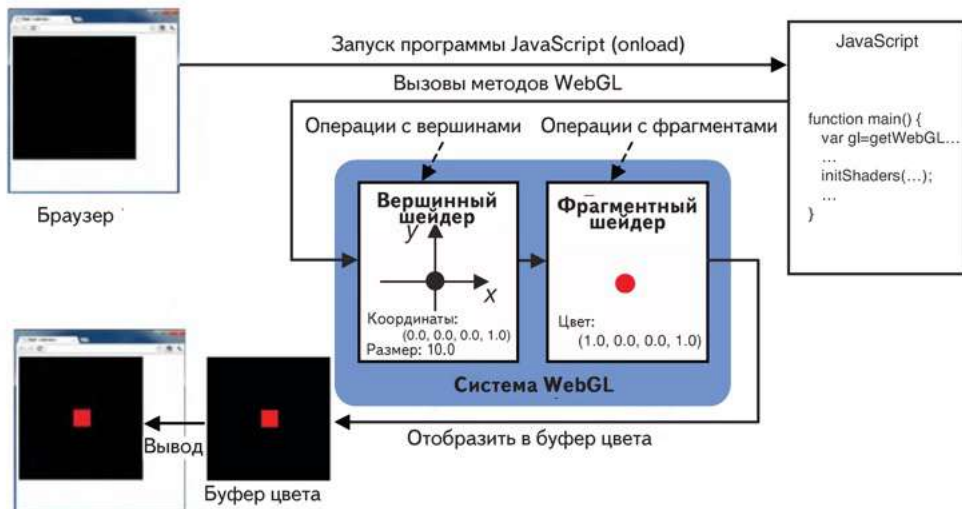


Рис. 2.10. Последовательность операций, от запуска программы JavaScript до отображения результатов в окне браузера

Слева на рис. 2.10 изображены два окна браузера. Это одно и то же окно в разные моменты времени. Вверху показано окно браузера перед выполнением программы JavaScript, а внизу – то же окно, но после выполнения программы. В момент вызова методов WebGL из программы на JavaScript, система WebGL выполнит вершинный и фрагментный шейдеры, чтобы вывести результат в буфер цвета. Это – этап очистки, то есть шаг с номером 4 на рис. 2.8, что приводился в описании оригинального примера `HelloCanvas`. После этого содержимое буфера цвета автоматически выводится в область рисования, определяемую элементом `<canvas>`.

Вы часто будете видеть этот рисунок далее в книге. Поэтому, для экономии места, мы будем приводить упрощенную его версию (см. рис. 2.11). Обратите внимание, что самый правый компонент на рисунке – это буфер цвета, а не окно браузера, потому что вывод содержимого буфера в окно производится автоматически.

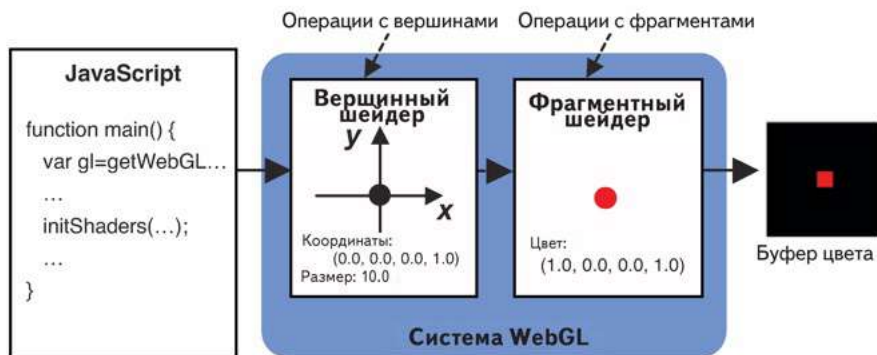


Рис. 2.11. Упрощенная версия рис. 2.10

Но, вернемся к нашему примеру. Цель его состоит в том, чтобы нарисовать точку размером 10 пикселей. Для этого используются два шейдера:

- вершинный шейдер определяет координаты точки и ее размер; в данном примере указаны координаты (0.0, 0.0, 0.0) и размер 10.0;
- фрагментный шейдер определяет цвет фрагментов точки; в данном примере выбран красный цвет (1.0, 0.0, 0.0, 1.0).

Структура WebGL-программы, использующей шейдеры

Опираясь на все, что мы узнали к данному моменту, рассмотрим содержимое файла `HelloPoint1.js` еще раз (см. листинг 2.5). Эта программа немного сложнее `HelloCanvas.js` (18 строк) и содержит 40 строк кода. Условно ее можно разделить на три части, как показано на рис. 2.12. Функция `main()` начинается со строки 15, а шейдеры определяются в строках со 2 по 13.

Собственно текст программы вершинного шейдера находится в строках с 4 по 7, а текст программы фрагментного шейдера – в строках с 11 по 13. Эти программы написаны на языке шейдеров, но включены в программу на JavaScript в виде строк, чтобы их можно было передать системе WebGL:

```
// Вершинный шейдер
void main() {
    gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
    gl_PointSize = 10.0;
}
// Фрагментный шейдер
void main() {
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

```
1 // HelloPint1.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4 'void main() {\n' +
5 '  gl_Position = vec4(0.0, 0.0, 0.0, 1.0);\n' +
6 '  gl_PointSize = 10.0;\n' +
7 '}\n';
8
9 // Фрагментный шейдер
10 var FSHADER_SOURCE =
11 'void main() {\n' +
12 '  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);\n' +
13 '}\n';
14
15 function main() {
16 // Получить ссылку на элемент <canvas>
17 var canvas = document.getElementById('webgl');
18
19 // Получить контекст отображения для WebGL
20 var gl = getWebGLContext(canvas);
21
22 ...
26 // Инициализировать шейдеры
27 if (!initShaders(gl, VSHADER_SOURCE, ...
28 ...
32 // Указать цвет для очистки <canvas>
33 gl.clearColor(0.0, 0.0, 0.0, 1.0);
34
35 ...
35 // Очистить <canvas>
36 gl.clear(gl.COLOR_BUFFER_BIT);
37
38 // Нарисовать точку
39 gl.drawArrays(gl.POINTS, 0, 1);
40 }
```

Вершинный шейдер
(язык: GLSL ES)

Фрагментный шейдер
(язык: GLSL ES)

Основная программа
(язык: JavaScript)

Рис. 2.12. Типичная структура WebGL-программы со встроенными шейдерами

Как вы уже знаете из главы 1, шейдеры пишутся на языке OpenGL ES (GLSL ES), который имеет некоторое сходство с языком C. Наконец-то на сцену вышел GLSL ES! Подробнее о языке шейдеров GLSL ES рассказывается в главе 6, «Язык шейдеров OpenGL ES (GLSL ES)», тем не менее, в этих первых примерах код на языке шейдеров настолько просто, что должен быть понятен любому, мало-мальски знакомому с языком C или JavaScript.

Так как эти программы должны интерпретироваться как единые строки, все строки в определении шейдера объединяются в одну строку с помощью оператора `+`. Каждая строка заканчивается символом перевода строки `\n`, потому что при выводе сообщений об ошибках в шейдерах указываются номера строк. Эти номера

помогут быстро найти источник ошибки в коде и устранить его. Однако символ `\n` не является обязательным, и вы можете записывать шейдеры без него.

В строках 3 и 10 шейдеры сохраняются в переменных `VSHADER_SOURCE` и `FSHADER_SOURCE`:

```
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'void main() {\n' +
5   '  gl_Position = vec4(0.0, 0.0, 0.0, 1.0);\n' +
6   '  gl_PointSize = 10.0;\n' +
7   '}\n';
8
9 // Фрагментный шейдер
10 var FSHADER_SOURCE =
11   'void main() {\n' +
12   '  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);\n' +
13   '}\n';
```

Если вам интересно узнать, как загружать шейдеры из отдельных файлов, обращайтесь к приложению F, «Загрузка шейдеров из файлов».

Инициализация шейдеров

Прежде чем углубляться в исследование шейдеров, посмотрим, какие операции выполняет функция `main()`, которая определена в строке 15 (см. рис. 2.13). Последовательность операций, представленная на рис. 2.13, является типичной для большинства WebGL-приложений. Ту же самую последовательность вы будете видеть на протяжении всей оставшейся части книги.



Рис. 2.13. Последовательность операций, выполняемых WebGL-программой

Эта последовательность напоминает ту, что была изображена на рис. 2.8, за исключением третьего («Инициализировать шейдеры») и шестого («Нарисовать») шагов, которые являются новыми.

На третьем шаге («Инициализировать шейдеры») осуществляется подготовка и передача в систему WebGL шейдеров, объявленных в строках 3 и 10. Этот шаг выполняется с помощью вспомогательной функции `initShaders()`, которая определена в файле `cuon-util.js`. Это одна из функций, которые мы написали сами, специально для этой книги.

`initShaders(gl, vshader, fshader)`

Инициализирует шейдеры и передает их системе WebGL:

Параметры	<code>gl</code>	Контекст отображения.
	<code>vshader</code>	Вершинный шейдер (строка).
	<code>fshader</code>	Фрагментный шейдер (строка).
Возвращаемое значение	<code>true</code>	Шейдеры успешно инициализированы
	<code>false</code>	Инициализация не увенчалась успехом.

На рис. 2.14 показано, как вспомогательная функция `initShaders()` обрабатывает шейдеры. Подробно эта функция будет исследоваться в главе 9. А пока вам достаточно будет узнать, как осуществляется подготовка шейдеров к работе в системе WebGL.

Как можно видеть в верхней части рис. 2.14, система WebGL имеет два контейнера: один для вершинного шейдера и один для фрагментного шейдера. Фактически мы допустили некоторое упрощение, но на данном этапе это несущественно. Мы вернемся к тонкостям реализации WebGL в главе 10. По умолчанию эти контейнеры пусты. Чтобы подготовить строки с шейдерами к использованию в системе WebGL, необходимо каким-то образом передать эти строки в систему и сохранить их в соответствующих контейнерах. Эту операцию выполняет функция `initShaders()`. Обратите внимание, что шейдеры выполняются внутри системы WebGL, а не в контексте программы JavaScript.

В нижней части рис. 2.14 показано, что получается в результате выполнения `initShaders()`: шейдеры, что передаются функции `initShaders()` в виде строковых параметров, сохраняются в контейнерах WebGL и становятся доступными для использования. В нижней части рисунка схематически показано, что вершинный шейдер передает `gl_Position` и `gl_PointSize` фрагментному шейдеру, и только после присваивания значений этим переменным в вершинном шейдере выполняется фрагментный шейдер. В действительности, фрагментному шейдеру передаются фрагменты, которые генерируются после обработки этих значений. Подробно этот механизм описывается в главе 5, а пока вы можете рассматривать это как передачу атрибутов.

Здесь следует отметить, что *WebGL-приложения состоят из программы на JavaScript, выполняемой браузером, и программ-шейдеров, выполняемых системой WebGL.*

Теперь, после исчерпывающего описания второго шага «Инициализировать шейдеры» (рис. 2.13), вы готовы увидеть, как использовать шейдеры для рисова-

ния простой точки. Как уже упоминалось, для рисования точки нам необходимо определить три ее характеристики: координаты, размер и цвет:

- вершинный шейдер определяет координаты точки и ее размер; в данном примере точка имеет координаты (0.0, 0.0, 0.0) и размер 10.0;
- фрагментный шейдер определяет цвет точки; в данном примере точка имеет красный цвет (1.0, 0.0, 0.0, 1.0).

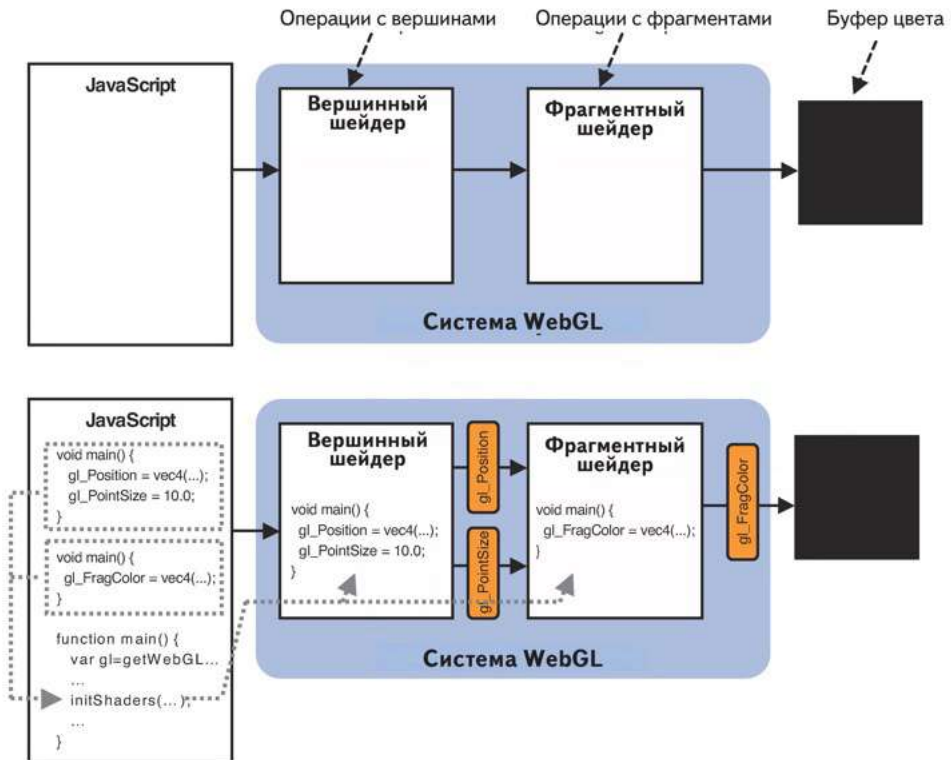


Рис. 2.14. Как действует функция `initShaders()`

Вершинный шейдер

Теперь приступим к исследованию вершинного шейдера, объявленного в `HelloPoint1.js` (см. листинг 2.5), который определяет координаты и размер точки:

```
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'void main() {\n' +
5     '  gl_Position = vec4(0.0, 0.0, 0.0, 1.0);\n' +
6     '  gl_PointSize = 10.0;\n' +
7   '}\n';
```

Собственно вершинный шейдер начинается в строке 4 и содержит определение единственной функции `main()`, напоминающее определения аналогичных функций в языках, подобных C. Ключевое слово `void` перед именем `main()` указывает, что эта функция не имеет возвращаемого значения. Кроме того, отсутствует возможность передачи каких-либо аргументов функции `main()`.

Так же, как и в языке JavaScript, для присваивания значений переменным, в шейдере можно использовать оператор `=`. В строке 5 выполняется присваивание координат точки переменной `gl_Position`, а в строке 6 – присваивание размера точки переменной `gl_PointSize`. Это – встроенные переменные, доступные только внутри вершинного шейдера и имеющие специальное значение: `gl_Position` определяет координаты вершины (в данном случае – координаты точки), а `gl_PointSize` определяет размеры точки (см. табл. 2.2).

Таблица 2.2. Встроенные переменные, доступные в вершинном шейдере

Тип и имя переменной	Описание
<code>vec4 gl_Position</code>	Определяет координаты вершины
<code>float gl_PointSize</code>	Определяет размер точки (в пикселях)

Отметьте, что вершинные шейдеры всегда должны присваивать значение переменной `gl_Position`. Если этого не сделать, поведение шейдера (в зависимости от реализации) может оказаться отличным от ожидаемого. Присваивать значение переменной `gl_PointSize`, напротив, требуется только при рисовании точек, и только если размер точки отличается от значения по умолчанию 1.0.

Тем, кто в основном знаком только с языком JavaScript, может показаться немного странным присутствие «типа» в табл. 2.2. В отличие от JavaScript, язык GLSL ES является «типизированным»; в том смысле, что он требует от программиста указывать типы данных, хранимых в переменных. Примерами типизированных языков могут служить C и Java. Благодаря наличию «типа» система легко может понять, данные какого типа могут храниться в переменных, и оптимизировать их обработку, опираясь на эту информацию. В табл. 2.3 перечислены «типы», поддерживаемые языком GLSL ES и используемые в этом разделе.

Таблица 2.3. Типы данных в языке GLSL ES

Тип	Описание				
float	Вещественное число (число с плавающей точкой).				
vec4	Вектор с четырьмя вещественными числами				
<table><tr><td>float</td><td>float</td><td>float</td><td>float</td></tr></table>		float	float	float	float
float	float	float	float		

Обратите внимание, что при попытке присвоить переменной данные несоответствующего типа, генерируется ошибка. Например, переменная `gl_PointSize` имеет тип `float`, соответственно ей можно присвоить только вещественное значение. То есть, если изменить строку 6


```
gl_PointSize = 10.0;
```

на

```
gl_PointSize = 10;
```

будет сгенерирована ошибка, просто потому, что число 10 интерпретируется как целочисленное значение, в отличие от числа 10.0, которое в языке GLSL ES считается вещественным значением.

Встроенная переменная `gl_Position`, определяющая координаты вершины, имеет тип `vec4` – вектор, состоящий из четырех вещественных значений. Однако, все, что у нас имеется – это три отдельных значения (0.0, 0.0, 0.0), представляющих координаты X, Y и Z. Поэтому нужен какой-то механизм, с помощью которого можно было бы преобразовать их в вектор типа `vec4`. К счастью такой механизм имеется – встроенная функция `vec4()`, которая преобразует свои аргументы в вектор `vec4` и возвращает его, то есть как раз то, что нам нужно!

vec4 vec4(v0, v1, v2, v3)

Конструирует объект типа `vec4` из четырех значений `v0`, `v1`, `v2` и `v3`

Параметры `v0, v1, v2, v3` Вещественные числа.

Возвращаемое значение Объект `vec4`, включающий значения `v0`, `v1`, `v2`, `v3`

В данном примере функция `vec4()` используется в строке 5:

```
gl_Position = vec4(0.0, 0.0, 0.0, 1.0);
```

Обратите внимание, что в четвертом элементе вектора, который присваивается переменной `gl_Position`, содержится значение 1.0. Такие четырехкомпонентные координаты называют **однородными координатами** (homogeneous coordinate) (см. врезку ниже) и часто используются в компьютерной графике для эффективной обработки трехмерной информации. Несмотря на то, что однородные координаты являются четырехкомпонентными, если четвертый компонент однородной координаты равен 1.0, такая координата соответствует той же позиции, которая может быть описана аналогичной трехкомпонентной координатой. То есть, определяя координаты вершины, указывайте в четвертом компоненте значение 1.0.

Однородные координаты

Однородные координаты (homogeneous coordinates) записываются в форме: (x, y, z, w) . Однородные координаты эквивалентны трехмерным координатам $(x/w, y/w, z/w)$. То есть, передавая в компоненте w значение 1.0, однородные координаты можно использовать как трехмерные координаты. Значение компонента w всегда должно быть больше или равно нулю. При приближении значения w к нулю, другие координаты стремятся к бесконечности. То есть, в системе однородных координат существует понятие бесконечности. Однородные координаты позволяют выразить преобразования вершин, которые описываются в следующей главе, через умножение матрицы на вектор координат. Эти координаты часто используются в качестве внутреннего представления вершин в системах трехмерной графики.

Фрагментный шейдер

После определения координат и размера точки необходимо также с помощью фрагментного шейдера определить ее цвет. Как описывалось выше, **фрагмент** – это пиксель, отображаемый на экране, который, технически, определяется позицией цветом и другой информацией.

Фрагментный шейдер – это программа, обрабатывающая данную информацию в процессе подготовки фрагмента к отображению на экране. Взгляните еще раз на определение фрагментного шейдера в файле `HelloPoint1.js` (см. листинг 2.5) – он так же как вершинный шейдер начинает выполнение с функции `main()`:

```
9 // Фрагментный шейдер
10 var FSHADER_SOURCE =
11     'void main() {\n' +
12     '   gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);\n' +
13     '}\n';
```

Задача шейдера состоит в том, чтобы установить цвет для каждого фрагмента (строка 12). Переменная `gl_FragColor` – это встроенная переменная, доступная только во фрагментном шейдере; она определяет цвет фрагмента, как показано в табл. 2.4.

Таблица 2.4. Встроенное значение, доступное во фрагментном шейдере

Тип и имя переменной	Описание
<code>vec4 gl_FragColor</code>	Определяет цвет фрагмента (в формате RGBA).

Когда мы присваиваем значение цвета встроенной переменной, фрагмент отображается с использованием этого цвета. Так же как координаты в вершинном шейдере, значение цвета имеет тип `vec4` и состоит из четырех вещественных чисел – значений RGBA. В данном примере точка будет окрашена в красный цвет, потому что переменной присваивается значение `(1.0, 0.0, 0.0, 1.0)`.

Операция рисования

После установки шейдеров остается только выполнить операцию рисования фигуры – в нашем случае точки. Как и прежде, нужно сначала очистить область рисования, подобно тому, как это делалось в примере `HelloCanvas.js`. После очистки области рисования можно нарисовать точку с помощью `gl.drawArrays()`, как показано в строке 39:

```
39 gl.drawArrays(gl.POINTS, 0, 1);
```

`gl.drawArrays()` – очень мощная функция, которая способна рисовать различные простейшие фигуры, как описывается ниже.

gl.drawArrays(mode, first, count)

Выполняет вершинный шейдер, чтобы нарисовать фигуры, определяемые параметром `mode`.

Параметры	<code>mode</code>	Определяет тип фигуры. Допустимыми значениями являются следующие константы: <code>gl.POINTS</code> , <code>gl.LINES</code> , <code>gl.LINE_STRIP</code> , <code>gl.LINE_LOOP</code> , <code>gl.TRIANGLES</code> , <code>gl.TRIANGLE_STRIP</code> и <code>gl.TRIANGLE_FAN</code> .
	<code>first</code>	Определяет номер вершины, с которой должно начинаться рисование (целое число).
	<code>count</code>	Определяет количество вершин (целое число).
Возвращаемое значение	нет	
Ошибки	<code>INVALID_ENUM</code>	В параметре <code>mode</code> передано значение, не являющееся ни одной из констант, перечисленных выше
	<code>INVALID_VALUE</code>	Отрицательное значение <code>first</code> и/или <code>count</code> .

В данном примере мы рисуем точку, поэтому в первом параметре `mode` передается значение `gl.POINTS`. Во втором параметре передается 0, потому что рисование начинается с первой вершины. В третьем параметре `count` передается 1, потому что программа рисует только одну точку.

Теперь, когда программа вызовет `gl.drawArrays()`, вершинный шейдер выполнится `count` раз, по одному разу для каждой вершины. В данном примере шейдер выполняется только один раз (параметр `count` имеет значение 1), потому что определена только одна вершина: наша точка. При вызове шейдера выполняется его функция `main()` – строка за строкой, – в результате чего переменной `gl_Position` присваивается значение (0.0, 0.0, 0.0, 1.0) (строка 5) и переменной `gl_PointSize` – значение 10.0 (строка 6).

Вслед за вершинным шейдером вызывается фрагментный шейдер и выполняется его функция `main()`, которая в этом примере присваивает переменной `gl_FragColor` значение, соответствующее красному цвету (строка 12). Как результат, в позиции с координатами (0.0, 0.0, 0.0, 1.0) выводится красная точка размером 10 пикселей, то есть, в центре области рисования (см. рис. 2.15).

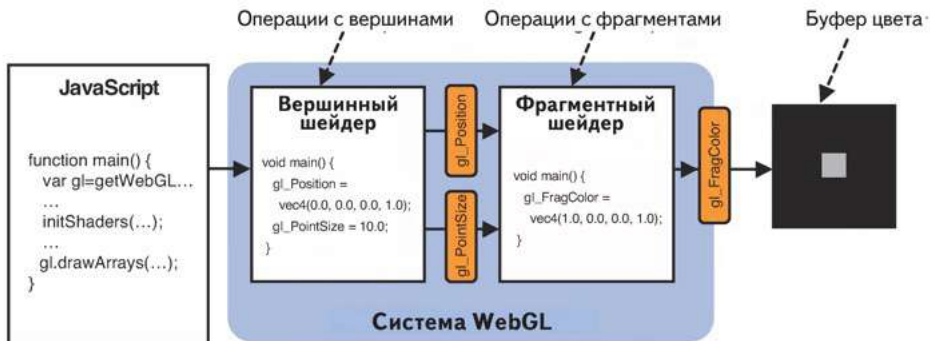


Рис. 2.15. Поведение шейдеров

К данному моменту у вас должно сложиться представление о назначении вершинного и фрагментного шейдеров и об особенностях их работы. В оставшейся части главы мы закрепим это представление, рассмотрев еще несколько примеров, что поможет вам лучше понять суть WebGL и шейдеров. Однако перед этим давайте посмотрим, как WebGL описывает позиции фигур с применением системы координат.

Система координат WebGL

Поскольку система WebGL работает с трехмерной графикой, она использует трехмерную систему координат с тремя осями: X, Y и Z. Эта система координат проста и понятна, потому что окружающий нас мир тоже имеет три измерения: ширину, высоту и глубину. В любой системе координат направление осей играет важную роль. Вообще, в WebGL, когда вы сидите перед плоскостью экрана монитора, ось X простирается по горизонтали, (слева направо), ось Y – по вертикали (снизу вверх) и ось Z – в направлении от плоскости экрана к пользователю (слева на рис. 2.16). Глаз пользователя находится в начале координат (0.0, 0.0, 0.0) и смотрит вдоль оси Z, в сторону отрицательных значений – за плоскость экрана (справа на рис. 2.16). Такая система координат называется также **правосторонней**, потому что ее можно изобразить пальцами правой руки (см. рис. 2.17), и она же обычно используется при работе с WebGL. На протяжении всей книги мы будем пользоваться именно правосторонней системой координат, как принятой по умолчанию в WebGL. Однако вы должны помнить, что в действительности все гораздо сложнее. Фактически система координат в WebGL не является ни правосторонней, ни левосторонней. Подробнее об этом рассказывается в приложении D, «WebGL/OpenGL: лево- или правосторонняя система координат?», но на данный момент вы можете считать систему координат в WebGL правосторонней.

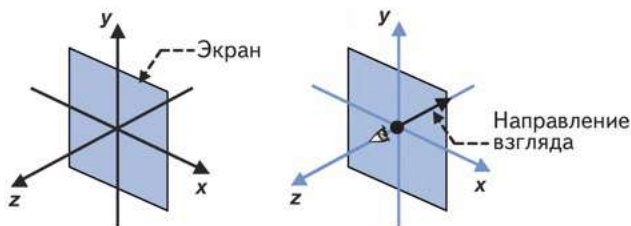


Рис. 2.16. Система координат в WebGL

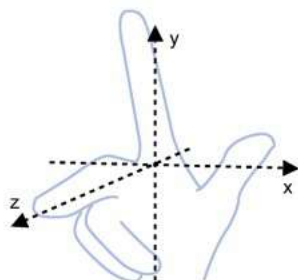


Рис. 2.17. Правосторонняя система координат

Как вы уже знаете, область рисования, что определяется элементом `<canvas>`, имеет свою систему координат, отличную от системы координат в WebGL, поэтому необходим некоторый механизм отображения одной системы в другую. По умолчанию, как показано на рис. 2.18, WebGL выполняет такое отображение следующим образом:

- центр области рисования `<canvas>`: $(0.0, 0.0, 0.0)$;
- две вертикальные границы области рисования `<canvas>`: $(-1.0, 0.0, 0.0)$ и $(1.0, 0.0, 0.0)$;
- две горизонтальные границы области рисования `<canvas>`: $(0.0, -1.0, 0.0)$ и $(0.0, 1.0, 0.0)$.

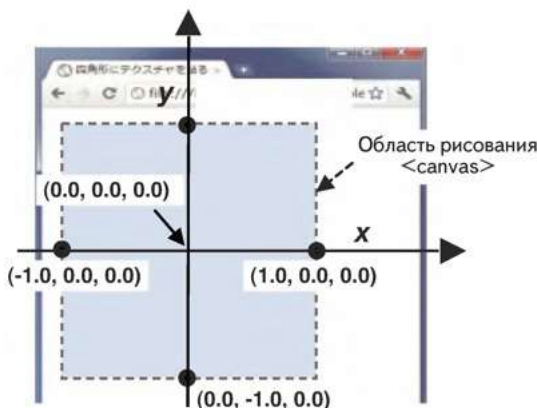


Рис. 2.18. Область рисования `<canvas>` и система координат WebGL

Как уже отмечалось, такое отображение выполняется по умолчанию. Однако, есть возможность использовать другую систему координат, о которой рассказывается ниже, но пока мы будем пользоваться системой координат по умолчанию. Кроме того, чтобы не отвлекать ваше внимание от исследования основной функциональности WebGL, в примерах программ будут использоваться только оси X и Y. То есть, вплоть до главы 7, координата глубины (ось Z) всегда будет иметь значение 0.0.

Эксперименты с примером программы

Для начала попробуем изменить координаты точки в строке 5, чтобы получить более ясное представление о системе координат WebGL. Например, давайте изменим координату X, подставив значение 0.5 вместо 0.0, как показано ниже:

```
5 ' gl_Position = vec4(0.5, 0.0, 0.0, 1.0);\n' +
```

Сохраните измененный файл `HelloPoint1.js` и щелкните на кнопке **Reload** (Обновить) браузера, чтобы перезагрузить его. Вы должны увидеть, что точка сместилась и теперь находится в правой половине области рисования (см. рис. 2.19, слева).

Теперь изменим координату Y, чтобы сместить точку вверх, как показано ниже:

```
5 ' gl_Position = vec4(0.0, 0.5, 0.0, 1.0);\n' +
```

Сохраните измененный файл `HelloPoint1.js` и перезагрузите его. На этот раз вы должны увидеть, что точка находится в верхней половине области рисования (см. рис. 2.19, справа).

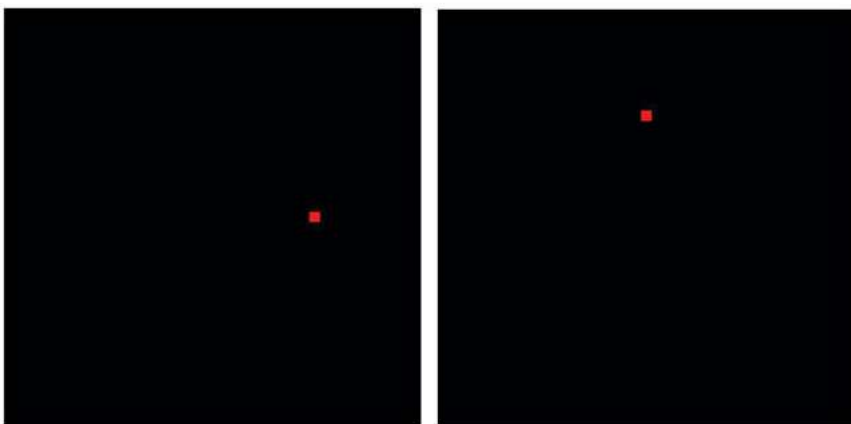


Рис. 2.19. Изменение местоположения точки

Проведем еще один эксперимент: попробуем изменить цвет точки с красного на зеленый, изменив строку 12, как показано ниже:

```
12  ' gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);\n'
```

Завершим этот раздел кратким подведением итогов. Вы познакомились с двумя основными разновидностями шейдеров – вершинными и фрагментными – и увидели, несмотря на то, что они написаны на собственном языке, как эти шейдеры выполняются из программного кода на JavaScript. Вы также узнали, что WebGL-приложения, использующие шейдеры, имеют ту же структуру, что и другие WebGL-приложения. Главный вывод данного раздела состоит в том, что WebGL-программы фактически являются программами на JavaScript, включающими программы-шейдеры.

Те из вас, кто имеет опыт использования библиотеки OpenGL, могут заметить, что что-то в наших примерах отсутствует – здесь нет кода, осуществляющего перестановку рабочего и фоновых буферов цвета. Одна из важнейших особенностей WebGL как раз и заключается в отсутствии такой необходимости. За дополнительной информацией обращайтесь к приложению А, «В WebGL нет необходимости использовать рабочий и фоновый буферы».

Рисование точки (версия 2)

В предыдущем разделе мы исследовали порядок рисования точки и необходимые для этого функции шейдеров. Теперь, когда вы получили некоторое представление об основных особенностях WebGL-программ, мы займемся исследованием механизма передачи данных между JavaScript и шейдерами. Программа

HelloPoint1 всегда рисует точку в одном и том же месте, жестко определенном в вершинном шейдере. Это делает пример проще для понимания, но лишает его гибкости. В этом разделе вы увидите, как WebGL-программа может передавать координаты вершины из JavaScript в вершинный шейдер, и затем рисовать точку в новом местоположении. Эта новая программа называется HelloPoint2, и хотя она воспроизводит тот же результат, что и программа HelloPoint1, достигается он более гибким способом, который мы будем использовать в будущих примерах.

Использование переменных-атрибутов

Наша цель – передать координаты точки из программного кода на JavaScript в вершинный шейдер. Сделать это можно двумя способами: с помощью переменных-атрибутов (переменных со спецификатором `attribute`) и `uniform`-переменных (переменных со спецификатором `uniform`) (см. рис. 2.20). Выбор того или иного спецификатора зависит от природы данных. Переменная-атрибут передает данные, уникальные для каждой вершины, тогда как `uniform`-переменная передает одни и те же данные для всех вершин. В данном примере мы будем использовать переменную-атрибут, потому что обычно все вершины имеют уникальные координаты.

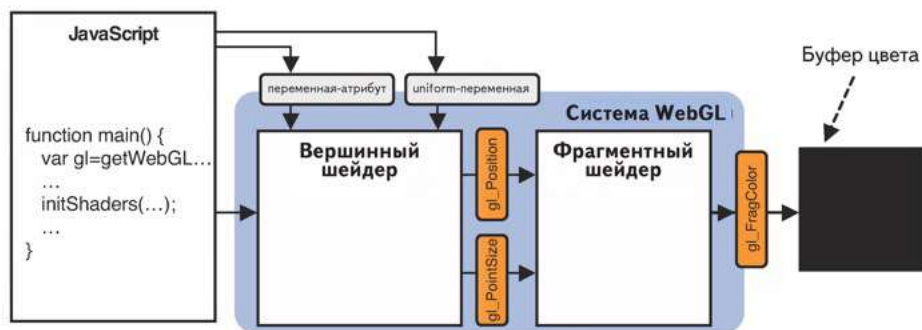


Рис. 2.20. Два способа передачи данных в вершинный шейдер

Переменные-атрибуты – это переменные GLSL ES, используемые для передачи данных из внешнего мира в вершинные шейдеры, и единственные, доступные вершинным шейдерам.

Чтобы задействовать переменные-атрибуты, необходимо выполнить следующие три шага:

1. объявить переменную-атрибут в вершинном шейдере;
2. присвоить встроенной переменной `gl_Position` значение переменной-атрибута;
3. присвоить данные переменной-атрибуту.

Рассмотрим пример программы, чтобы увидеть, как реализуются эти шаги.

Пример программы (HelloPoint2.js)

Программа HelloPoint2 (см. листинг 2.6) рисует точку в координатах, определяемых программой на JavaScript.

Листинг 2.6. HelloPoint2.js

```
1 // HelloPoint2.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'void main() {\n' +
6     '  gl_Position = a_Position;\n' +
7     '  gl_PointSize = 10.0;\n' +
8     '}\n';
9
10 // Фрагментный шейдер
    ... фрагмент вырезан, так как он идентичен фрагменту в HelloPoint1.js
15
16 function main() {
17   // Получить ссылку на элемент <canvas>
18   var canvas = document.getElementById('webgl');
19
20   // Получить контекст отображения для WebGL
21   var gl = getWebGLContext(canvas);
22   ...
23
24   // Инициализировать шейдеры
25   if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
26     ...
27   }
28
29   // Получить ссылку на переменную-атрибут
30   var a_Position = gl.getAttributeLocation(gl.program, 'a_Position');
31   if (a_Position < 0) {
32     console.log('Failed to get the storage location of a_Position');
33     return;
34   }
35
36   // Сохранить координаты в переменной-атрибуте
37   gl.vertexAttrib3f(a_Position, 0.0, 0.0, 0.0);
38
39   // Указать цвет для очистки области рисования <canvas>
40   gl.clearColor(0.0, 0.0, 0.0, 1.0);
41
42   // Очистить <canvas>
43   gl.clear(gl.COLOR_BUFFER_BIT);
44
45   // Нарисовать точку
46   gl.drawArrays(gl.POINTS, 0, 1);
47 }
```

Как видите, объявление переменной-атрибута находится внутри шейдера, в строке 4:

```
4  'attribute vec4 a_Position;\n' +
```

Ключевое слово `attribute` в этой строке называется **спецификатором класса хранения** и указывает, что следующая за ним переменная (в данном случае `a_Position`) является переменной-атрибутом. Эта переменная должна быть объявлена глобальной, потому что данные в нее записываются за пределами шейдера. Объявление должно следовать стандартному шаблону: `<Storage Qualifier> <Type> <Variable Name>`, как показано на рис. 2.21.

Спецификатор
класса хранения Тип Имя переменной

attribute vec4 a_Position;

Рис. 2.21. Объявление переменной-атрибута

В строке 4 переменная `a_Position` объявляется как переменная-атрибут с типом `vec4`, потому что, как было показано в табл. 2.2, ее значение будет присваиваться встроенной переменной `gl_Position`, которая всегда имеет тип `vec4`.

Обратите внимание, что в этой книге мы будем следовать соглашениям об именовании переменных, в соответствии с которыми имена переменных-атрибутов должны начинаться с префикса `a_`, а имена `uniform`-переменных – с префикса `u_`, чтобы при просмотре программного кода было проще определять типы переменных по их именам. Безусловно, вы можете использовать собственные соглашения об именовании, но мы посчитали наши соглашения простыми и понятными.

После объявления переменной `a_Position`, ее значение присваивается встроенной переменной `gl_Position` (строка 6):

```
6  ' gl_Position = a_Position;\n' +
```

На этом подготовку шейдера к приему данных извне можно считать законченной. Следующий шаг – передача данных в переменную-атрибут из программы на JavaScript.

Получение ссылки на переменную-атрибут

Как вы уже видели выше, вершинный шейдер передается в систему WebGL с помощью вспомогательной функции `initShaders()`. Когда осуществляется передача вершинного шейдера в WebGL, система анализирует его, обнаруживает переменную-атрибут и выделяет область памяти для хранения ее значения. Чтобы передать данные в переменную `a_Position`, находящуюся внутри шейдера, необходимо запросить у системы WebGL местоположение этой переменной (ссылку на нее), вызвав метод `gl.getAttribLocation()`, как показано в строке 34:

```
33  // Получить ссылку на переменную-атрибут
34  var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
```



```

35 if (a_Position < 0) {
36     console.log('Fail to get the storage location of a_Position');
37     return;
38 }

```

Первый аргумент этого метода определяет **объект программы**, хранящий вершинный и фрагментный шейдеры. Подробнее об объекте программы будет рассказываться в главе 8, а пока мы просто будем использовать объект `gl.program` в качестве аргумента. Отметьте, что объект `gl.program` можно использовать только после вызова функции `initShaders()`, потому что `initShaders()` присваивает этот объект переменной. Второй параметр определяет имя переменной-атрибута (в данном случае `a_Position`) ссылку на которую требуется получить.

Этот метод возвращает ссылку на указанную переменную-атрибут, которая затем сохраняется в JavaScript-переменной `a_Position`, в строке 34. И снова, чтобы упростить чтение кода, при выборе имен переменных JavaScript мы следуем тому же соглашению об именовании. Разумеется, вы можете давать своим переменным любые другие имена.

Ниже приводится описание `gl.getAttribLocation()`:

`gl.getAttribLocation(program, name)`

Возвращает ссылку на переменную-атрибут, определяемую параметром `name`.

Параметры	<code>program</code>	Объект программы, хранящий вершинный и фрагментный шейдеры.
	<code>name</code>	Определяет имя переменной-атрибута, ссылку на которую требуется получить.
Возвращаемое значение	целое число большее или равное 0	Ссылка на указанную переменную-атрибут.
	-1	Указанная переменная-атрибут не найдена или ее имя начинается с зарезервированного префикса <code>gl_</code> или <code>webgl_</code> .
Ошибки	<code>INVALID_OPERATION</code>	Объект программы <code>program</code> не был скомпонован (см. главу 9).
	<code>INVALID_VALUE</code>	Длина имени <code>name</code> больше максимально возможной длины (256 символов по умолчанию).

Присваивание значения переменной-атрибуту

После получения ссылки на переменную-атрибут, ей необходимо присвоить значение. Эта операция выполняется в строке 41, с помощью метода `gl.vertexAttrib3f()`.

```

40 // Сохранить координаты в переменной-атрибуте
41 gl.vertexAttrib3f(a_Position, 0.0, 0.0, 0.0);

```

Ниже приводится описание `gl.vertexAttrib3f()`.

gl.vertexAttrib3f(location, v0, v1, v2)

Присваивает данные (v0, v1 и v2) переменной-атрибуту, определяемой аргументом location.

Параметры	location	Ссылка на переменную-атрибут, которой требуется присвоить указанное значение.
	v0	Значение, используемое как первый элемент для переменной-атрибута.
	v1	Значение, используемое как второй элемент для переменной-атрибута.
	v2	Значение, используемое как третий элемент для переменной-атрибута.
Возвращаемое значение	нет	
Ошибки	INVALID_OPERATION	Объект программы <code>program</code> не был скомпонован (см. главу 9).
	INVALID_VALUE	Ссылка <code>location</code> больше или равна максимально возможному значению для ссылок на переменные-атрибуты (8 по умолчанию).

В первом аргументе методу передается ссылка, полученная в результате вызова `gl.getAttribLocation()` в строке 34. Во втором, третьем и четвертом аргументах передаются вещественные значения, представляющие координаты X, Y и Z точки. После вызова метода эти три значения будут записаны в переменную-атрибут `a_Position`, которая была объявлена в строке 4, внутри вершинного шейдера. На рис. 2.22 показано, как происходит получение ссылки на переменную-атрибут и последующая запись значения по ссылке.

Затем значение переменной `a_Position` присваивается встроенной переменной `gl_Position` в строке 6, внутри вершинного шейдера. В результате этого координаты X, Y и Z, что передаются из JavaScript через переменную-атрибут в шейдер, оказываются в переменной `gl_Position`. То есть, данная программа производит тот же результат, что и программа `HelloPoint1`, где координаты точки определяются переменной `gl_Position`. Однако на этот раз значение переменной `gl_Position` устанавливается динамически, из JavaScript:

```

4      'attribute vec4 a_Position;\n' +
5      'void main() {\n' +
6      '    gl_Position = a_Position;\n' +
7      '    gl_PointSize = 10.0;\n' +
8      '}\n';

```

В завершение выполняется очистка области рисования `<canvas>` вызовом `gl.clear()` (строка 47) и рисование точки вызовом `gl.drawArrays()` (строка 50), точно так же, как в программе `HelloPoint1.js`.

И последнее замечание. Как можно видеть в листинге 2.6, переменная `a_Position` объявлена в строке 4 с типом `vec4`. Однако в вызов метода `gl.vertexAttrib3f()`

(строка 41) передается только три значения (координаты X, Y и Z), а не четыре. Дело в том, что этот метод автоматически подставит 1.0 вместо отсутствующего четвертого значения (см. рис. 2.23). Как было показано выше, в определении цвета четвертое значение 1.0, которое используется по умолчанию, обеспечит полную непрозрачность, а в определении координат четвертое значение 1.0 обеспечит отображение трехмерных координат в однородные координаты, то есть, по сути, метод подставляет «безопасное» четвертое значение.

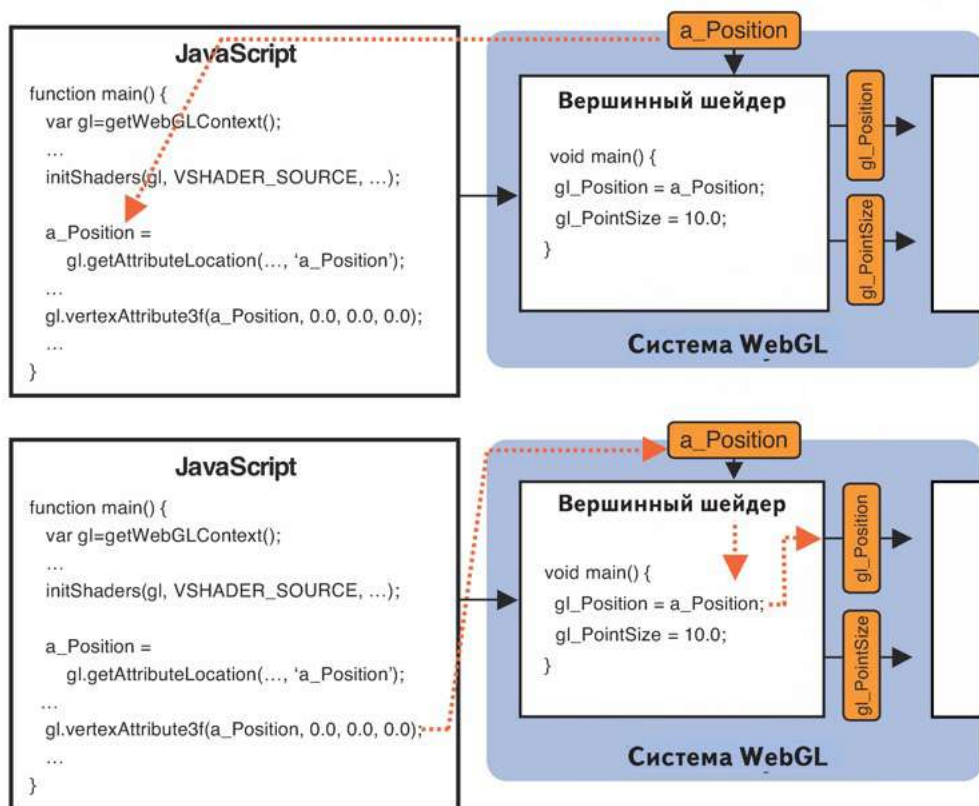


Рис. 2.22. Получение ссылки на переменную-атрибут и запись значения в эту переменную

Семейство методов `gl.vertexAttrib3f()`

Метод `gl.vertexAttrib3f()` является частью семейства методов, позволяющих устанавливать значения отдельных или всех компонентов переменной-атрибута. Метод `gl.vertexAttrib1f()` устанавливает единственное значение (`v0`), метод `gl.vertexAttrib2f()` — два значения (`v0` и `v1`) и метод `gl.vertexAttrib4f()` — четыре значения (`v0`, `v1`, `v2` и `v3`).

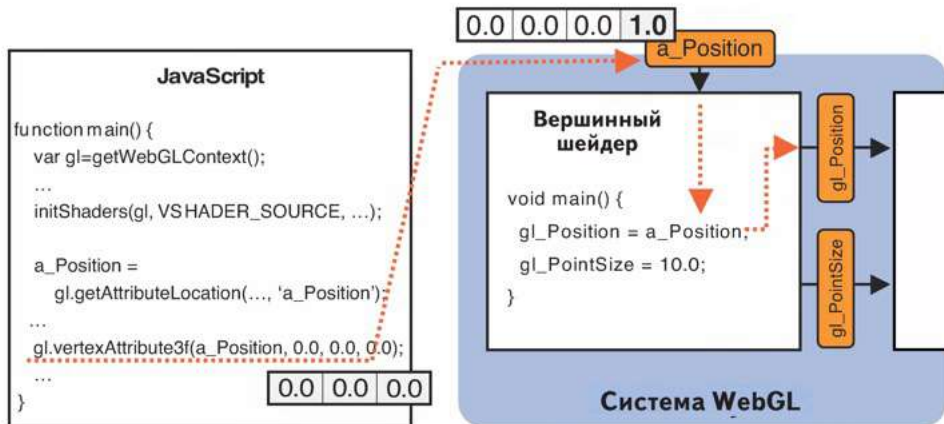


Рис. 2.23. Автоматическая подстановка недостающего значения

```
gl.vertexAttrib1f(location, v0)
gl.vertexAttrib2f(location, v0, v1)
gl.vertexAttrib3f(location, v0, v1, v2)
gl.vertexAttrib4f(location, v0, v1, v2, v3)
```

Присваивают значение переменной-атрибуту по ссылке `location`. Метод `gl.vertexAttrib1f()` присваивает единственное значение и используется для изменения первого элемента вектора переменной-атрибута. Второму и третьему элементам вектора будут присвоены значения 0.0, а четвертому – значение 1.0. Аналогично действует метод `gl.vertexAttrib2f()`, который присваивает значения первым двум элементам, третьему присваивает значение 0.0, а четвертому – значение 1.0. Метод `gl.vertexAttrib3f()` присваивает значения первым трем элементам, а четвертому – значение 1.0. Метод `gl.vertexAttrib4f()` присваивает значения всем четырем элементам вектора.

Параметры	<code>location</code>	Ссылка на переменную-атрибут, которой требуется присвоить указанное значение.
	<code>v0, v1, v2 и v3</code>	Значения, которые должны быть присвоены первому, второму, третьему и четвертому элементам переменной-атрибута.
Возвращаемое значение	нет	
Ошибки	<code>INVALID_VALUE</code>	Ссылка <code>location</code> больше или равна максимально возможному значению для ссылок на переменные-атрибуты (8 по умолчанию).

Существуют также «векторные» версии этих методов, принимающие вектор в качестве аргумента (см. главу 4). Их имена оканчиваются символом «v» (vector – вектор). Число в имени метода указывает на количество элементов в векторе. Например,

```
var position = new Float32Array([1.0, 2.0, 3.0, 1.0]);
gl.vertexAttrib4fv(a_Position, position);
```

В данном случае число 4 в имени метода указывает, что он принимает массив с четырьмя элементами.

Правила именования методов WebGL

Кому-то из вас может быть любопытно, что означает 3f в имени `gl.vertexAttrib3f()`. Имена методов в WebGL выбирались, исходя из имен функций, определяемых спецификацией в OpenGL ES 2.0, которая, как вы теперь знаете, является основой для WebGL. Имена функций в OpenGL включают три компонента: <базовое имя функции> <число параметров> <тип параметра>. Имена всех методов WebGL включают те же три компонента: <базовое имя метода> <число параметров> <тип параметра>, как показано на рис. 2.24.

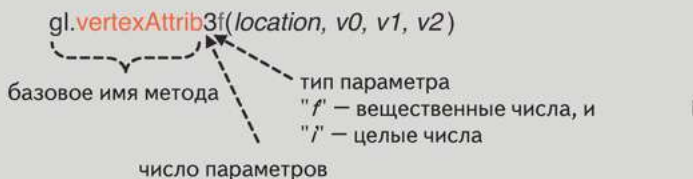


Рис. 2.24. Правила именования методов WebGL

Исходя из этой схемы, можно заключить, что метод `gl.vertexAttrib3f()` имеет базовое имя `vertexAttrib`, принимает 3 параметра вещественного типа (f — означает float, то есть вещественное число). Данный метод является WebGL-версией функции `glVertexAttrib3f()` в OpenGL. В качестве индикатора типа параметра может также использоваться символ i, который обозначает целое число (integer). Для обозначения всех методов, от `gl.vertexAttrib1f()` до `gl.vertexAttrib4f()`, можно использовать такую нотацию: `gl.vertexAttrib[1234]f()`.

Где квадратные скобки [] указывают, что на их месте должно быть указано одно из перечисленных чисел.

Когда в конце имени присутствует дополнительный символ v, это говорит о том, что метод принимает не отдельные значения, а вектор (массив) значений. В данном случае число в имени метода указывает на количество элементов в векторе.

```
var positions = new Float32Array([1.0, 2.0, 3.0, 1.0]);
gl.vertexAttrib4fv(a_Position, positions);
```

Эксперименты с примером программы

Теперь, когда у нас имеется программа, передающая координаты точки из кода на JavaScript в вершинный шейдер, можно попробовать поэкспериментировать с этими координатами. Например, если вы пожелаете отобразить точку в позиции с координатами (0.5, 0.0, 0.0), вы могли бы изменить программу, как показано ниже:

```
33 gl.vertexAttrib3f(a_Position, 0.5, 0.0, 0.0);
```

Для решения той же задачи можно было бы использовать другие методы семейства `gl.vertexAttrib3f()`:

```
gl.vertexAttrib1f(a_Position, 0.5);
gl.vertexAttrib2f(a_Position, 0.5, 0.0);
gl.vertexAttrib4f(a_Position, 0.5, 0.0, 0.0, 1.0);
```

Теперь, научившись пользоваться переменными-атрибутами, можно попробовать применить тот же подход к изменению размера точки из программы на JavaScript. Для этого нужно объявить новую переменную-атрибут, через которую в шейдер будет передаваться размер точки. Следуя соглашениям об именовании, принятым в этой книге, определим переменную `a_PointSize`. Как было показано в табл. 2.2, встроенная переменная `gl_PointSize` имеет тип `float`, соответственно переменная-атрибут так же должна иметь тип `float`:

```
attribute float a_PointSize;
```

Соответственно, вершинный шейдер должен определяться так:

```
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'attribute float a_PointSize; \n' +
6   'void main() {\n' +
7   '   gl_Position = a_Position;\n' +
8   '   gl_PointSize = a_PointSize;\n' +
9   '}\n';
```

Далее, нужно получить ссылку на переменную-атрибут `a_PointSize` и присвоить ей требуемое значение, для чего можно воспользоваться методом `gl.vertexAttrib1f()`. Так как переменная `a_PointSize` имеет тип `float`, метод `gl.vertexAttrib1f()` следует вызывать, как показано ниже:

```
33 // Получить ссылку на переменную-атрибут
34 var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
...
39 var a_PointSize = gl.getAttribLocation(gl.program, 'a_PointSize');
40 // Сохранить координаты в переменной-атрибуте
41 gl.vertexAttrib3f(a_Position, 0.0, 0.0, 0.0);
42 gl.vertexAttrib1f(a_PointSize, 5.0);
```

А теперь поэкспериментируйте с программой самостоятельно, чтобы лучше понять, как пользоваться переменными-атрибутами и как они действуют.

Рисование точки по щелчку мышью

Предыдущая программа `HelloPoint2` способна передавать координаты точки в вершинный шейдер из программного кода на JavaScript. Однако координаты все еще жестко «зашиты» в код, поэтому программа `HelloPoint2` мало чем отличается от программы `HelloPoint1`, в которой координаты были жестко «зашиты» в определение шейдера.

В этом разделе мы добавим еще гибкости и реализуем передачу в шейдер координат точки, где был выполнен щелчок мышью. На рис. 2.25 показан снимок с экрана программы `ClickedPoint`.³

³ © 2012 Марисуке Канья (Marisuke Kunnya)

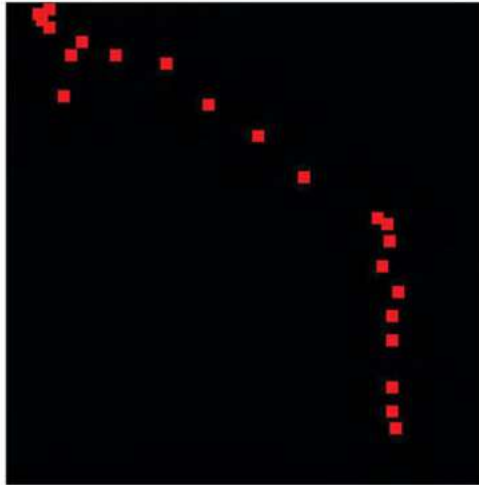


Рис. 2.25. ClickedPoint

Пример программы (ClickedPoints.js)

В листинге 2.7 приводится содержимое файла `ClickedPoints.js`. Для экономии места мы убрали фрагменты кода, которые совпадают с фрагментами в предыдущем примере и заменили их многоточиями.

Листинг 2.7. ClickedPoints.js

```
1 // ClickedPoints.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'void main() {\n' +
6     '  gl_Position = a_Position;\n' +
7     '  gl_PointSize = 10.0;\n' +
8   '}\n';
9
10 // Фрагментный шейдер
11 ...
12
13 function main() {
14   // Получить ссылку на элемент <canvas>
15   var canvas = document.getElementById('webgl');
16
17   // Получить контекст отображения для WebGL
18   var gl = getWebGLContext(canvas);
19   ...
20   // Инициализировать шейдеры
21   if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)){
22     ...
23   }
24 }
25
26 ...
27
28 ...
29
30 ...
31 }
32
```

```
33 // Получить ссылку на переменную-атрибут a_Position
34 var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
35 ...
40 // Зарегистрировать функцию (обработчик) для вызова по щелчку мышью
41 canvas.onmousedown = function(ev) { click(ev, gl, canvas, a_Position); };
42 ...
47 gl.clear(gl.COLOR_BUFFER_BIT);
48 }
49
50 var g_points = []; // Массив с координатами точек, где выполнялись щелчки
51 function click(ev, gl, canvas, a_Position) {
52     var x = ev.clientX; // координата X указателя мыши
53     var y = ev.clientY; // координата Y указателя мыши
54     var rect = ev.target.getBoundingClientRect();
55
56     x = ((x - rect.left) - canvas.width/2)/(canvas.width/2);
57     y = ((y - rect.top) - canvas.height/2)/(canvas.height/2);
58     // Сохранить координаты в массиве g_points
59     g_points.push(x); g_points.push(y);
60
61     // Очистить <canvas>
62     gl.clear(gl.COLOR_BUFFER_BIT);
63
64     var len = g_points.length;
65     for(var i = 0; i < len; i+=2) {
66         // Передать координаты щелчка в переменную a_Position
67         gl.vertexAttrib3f(a_Position, g_points[i], g_points[i+1], 0.0);
68
69         // Нарисовать точку
70         gl.drawArrays(gl.POINTS, 0, 1);
71     }
72 }
```

Регистрация обработчиков событий

Порядок работы этой программы в строках с 17 по 39 остался прежним, как в программе `HelloPoint2.js`. В этих строках программа получает контекст отображения WebGL, инициализирует шейдеры и затем получает ссылку переменной-атрибута. Основное отличие от программы `HelloPoint2.js` заключено в регистрации обработчика событий (строка 41) и в определении функции-обработчика `click()` (начиная со строки 51).

Обработчик события – это асинхронная функция обратного вызова, обрабатывающая события ввода от пользователя, такие как щелчки мышью или нажатия клавиш на клавиатуре. Возможность определения обработчиков позволяет создавать динамические веб-страницы и изменять их содержимое в соответствии с вводом пользователя. Чтобы задействовать обработчик, его нужно зарегистрировать (то есть, сообщить системе, что она должна вызывать функцию-обработчик при появлении указанного события). Элемент `<canvas>` поддерживает специальные свойства для регистрации обработчиков, которые и используются в данном примере.

Например, чтобы получить возможность обрабатывать события щелчков мышью, нужно присвоить свойству `onmousedown` элемента `<canvas>` ссылку на функцию, которая будет обрабатывать щелчки, как показано ниже:

```
40 // Зарегистрировать функцию (обработчик) для вызова по щелчку мышью
41 canvas.onmousedown = function(ev) { click(ev, gl, canvas, a_Position); };
```

Для регистрации обработчика в строке 41 используется инструкция `function()` `{ ... }`:

```
function(ev){ click(ev, gl, canvas, a_Position); }
```

Это – определение **анонимной функции**, то есть функции, не имеющей имени, что очень удобно, когда требуется определить функцию, не требующую уникального имени.

Для тех, кто не знаком с функциями подобного рода, объясним их суть на примере. В следующей строке программного кода определяется переменная `thanks`:

```
var thanks = function(){ alert(' Thanks a million!'); }
```

Эту переменную можно вызвать как функцию:

```
thanks(); // Выведет: 'Thanks a million!'
```

Как видите, переменная `thanks` может выступать в роли функции. Эти строки можно переписать иначе:

```
function thanks() { alert('Thanks a million!'); }
thanks(); // Выведет: 'Thanks a million!'
```

Но с какой целью здесь используется анонимная функция? Дело в том, что для рисования точки нам нужны три переменные: `gl`, `canvas` и `a_Position`. Эти три переменными являются локальными и определяются в функции `main()`. Но, когда возникает событие щелчка, браузер автоматически вызовет функцию, зарегистрированную в свойстве `onmousedown` элемента `<canvas>` с **единственным предопределенным параметром** (то есть, объектом события, содержащем информацию о щелчке). Обычно в программах на JavaScript сначала объявляется функция, а затем она регистрируется как обработчик события:

```
function mousedown(ev) { // Обработчик события: принимает один параметр 'ev'
    ...
}
...
canvas.onmousedown = mousedown; // Зарегистрировать 'mousedown' как обработчик
```

Однако, такая функция не сможет получить доступ к локальным переменным `gl`, `canvas` и `a_Position`. Анонимная функция в строке 41 решает эту проблему, так как будучи объявленной в области видимости этих переменных, может обращаться к ним:

```
41 canvas.onmousedown = function(ev) { click(ev, gl, canvas, a_Position); };
```


В этом случае, когда возникает событие щелчка мышью, сначала вызывается анонимная функция `function(ev)`, которая затем выполняет вызов `click(ev, gl, canvas, a_Position)` и передает локальные переменные, объявленные в `main()`. Кому-то такой прием может показаться немного сложным, но в действительности он дает большую гибкость и позволяет избежать необходимости использовать глобальные переменные, чтобы всегда хорошо. Пройдитесь по приведенному примеру еще раз, что до конца понять суть этого приема, потому что он часто будет использоваться в этой книге для регистрации обработчиков.

Обработка события щелчка мышью

Давайте посмотрим, что собственно делает функция `click()`. Она выполняет следующие действия:

1. Получает координаты указателя мыши в момент щелчка и сохраняет их в массиве.
2. Очищает `<canvas>`.
3. Для каждой пары координат в массиве рисует точку.

```
50 var g_points = []; // Массив с координатами точек, где выполнялись щелчки
51 function click(ev, gl, canvas, a_Position) {
52   var x = ev.clientX; // координата X указателя мыши
53   var y = ev.clientY; // координата Y указателя мыши
54   var rect = ev.target.getBoundingClientRect();
55
56   x = ((x - rect.left) - canvas.width/2)/(canvas.width/2);
57   y = (canvas.height/2 - (y - rect.top))/(canvas.height/2);
58   // Сохранить координаты в массиве g_points
59   g_points.push(x); g_points.push(y);
60
61   // Очистить <canvas>
62   gl.clear(gl.COLOR_BUFFER_BIT);
63
64   var len = g_points.length;
65   for(var i = 0; i < len; i+=2) {
66     // Передать координаты щелчка в переменную a_Position
67     gl.vertexAttrib3f(a_Position, g_points[i], g_points[i+1], 0.0);
68
69     // Нарисовать точку
70     gl.drawArrays(gl.POINTS, 0, 1);
71   }
72 }
```

Координаты указателя мыши в момент щелчка хранятся в объекте события, который передается браузером в аргументе `ev`. Извлечь координаты из объекта `ev` можно с обращением к свойствам `ev.clientX` и `ev.clientY`, как показано в строках 52 и 53. Однако эти координаты нельзя использовать непосредственно по двум причинам:

1. Координаты соответствуют положению указателя мыши в «клиентской области» окна браузера, а не в элементе `<canvas>` (см. рис. 2.26).



Рис. 2.26. Система координат клиентской области окна браузера и координаты элемента `<canvas>`

2. Система координат элемента `<canvas>` отличается от системы координат WebGL (см. рис. 2.27) – начало системы координат и направление оси Y не совпадают.

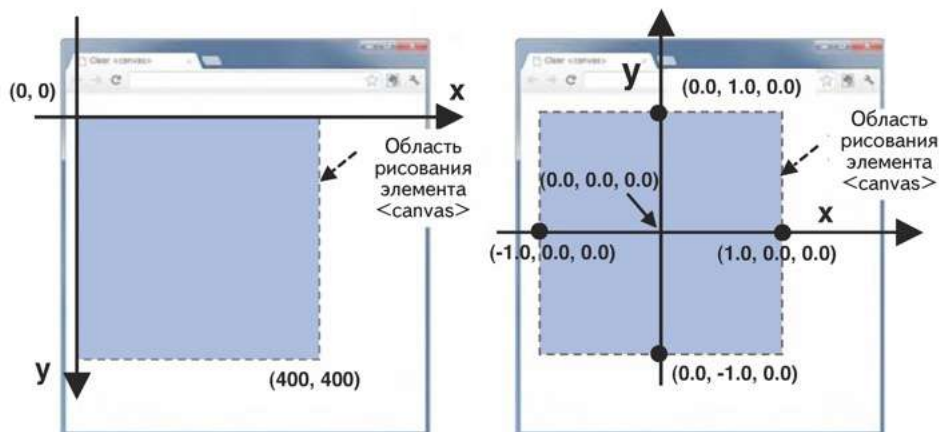


Рис. 2.27. Система координат элемента `<canvas>` (слева) и система координат WebGL в элементе `<canvas>` (справа)

Прежде всего нужно преобразовать координаты из системы координат клиентской области окна браузера в систему координат элемента `<canvas>`, а затем – в систему координат WebGL. Давайте посмотрим, как это делается.

Необходимые преобразования выполняются в строках 56 и 57 программы:

```

52  var x = ev.clientX;
53  var y = ev.clientY;
54  var rect = ev.target.getBoundingClientRect();
55  ...
56  x = ((x - rect.left) - canvas.width/2)/(canvas.width/2);
57  y = (canvas.height/2 - (y - rect.top))/(canvas.height/2);

```

В строке 54 выполняется преобразование координат из системы координат клиентской области окна браузера в систему координат элемента `<canvas>`. Значения `rect.left` и `rect.top` – это координаты верхнего левого угла `<canvas>` в клиентской области окна браузера (см. рис. 2.26). То есть, выражение $(x - \text{rect.left})$ в строке 56 и выражение $(y - \text{rect.top})$ в строке 57 смещают начало координат в позицию верхнего левого угла элемента `<canvas>`.

Далее нам нужно преобразовать координаты в элементе `<canvas>` в систему координат WebGL, как показано на рис. 2.27. Для этого требуется определить координаты центра элемента `<canvas>`. Получить высоту и ширину элемента `<canvas>` можно с помощью свойств `canvas.height` (в данном случае имеет значение 400) и `canvas.width` (так же имеет значение 400). Таким образом, центр элемента `<canvas>` будет иметь координаты $(\text{canvas.height}/2, \text{canvas.width}/2)$.

Далее нам нужно реализовать это преобразование путем смещения начала координат в центр элемента `<canvas>`, где находится начало системы координат WebGL. Необходимое преобразование выполняется с помощью выражений $((x - \text{rect.left}) - \text{canvas.width}/2)$ и $(\text{canvas.height}/2 - (y - \text{rect.top}))$.

Наконец, как показано на рис. 2.27, диапазон значений по оси X в элементе `<canvas>` изменяется от 0 до `canvas.width` (400), а диапазон значений по оси Y – от 0 до `canvas.height` (400). Но, так как диапазон значений по осям координат в WebGL изменяется от -1.0 до 1.0, на последнем шаге преобразования системы координат `<canvas>` в систему координат WebGL необходимо разделить координату X на `canvas.width/2`, а координату Y – на `canvas.height/2`. Это деление выполняется в строках 56 и 57.

В строке 59 преобразованные координаты указателя мыши сохраняются в массиве `g_points`, с помощью метода `push()`, который добавляет данные в конец массива:

```
59 g_points.push(x); g_points.push(y);
```

Итак, каждый раз, когда возникает событие щелчка, координаты указателя мыши добавляются в конец массива, как показано на рис. 2.28. (Длина массива при этом автоматически увеличивается.) Обратите внимание, что нумерация элементов массива начинается с 0, поэтому первый элемент доступен как `g_points[0]`.

содержимое массива `g_points []`

координата X 1-го щелчка	координата Y 1-го щелчка	координата X 2-го щелчка	координата Y 2-го щелчка	координата X 3-го щелчка	координата Y 3-го щелчка	...
<code>g_points[0]</code>	<code>g_points[1]</code>	<code>g_points[2]</code>	<code>g_points[3]</code>	<code>g_points[4]</code>	<code>g_points[5]</code>	

Рис. 2.28. Содержимое массива `g_points`

Кому-то из вас может быть интересно узнать, зачем хранить координаты всех щелчков, а не только самого последнего. Это обусловлено особенностями использования буфера цвета в WebGL. Напомним, как было показано на рис. 2.10, что при использовании системы WebGL рисование выполняется сначала в буфере

цвета, и лишь затем система отображает содержимое буфера на экране. После этого буфер инициализируется повторно и его содержимое утрачивается. (Таково поведение буфера по умолчанию, о чем будет рассказываться в следующем разделе.) То есть, чтобы программа могла нарисовать все точки, соответствующие предыдущим щелчкам, необходимо хранить координаты всех щелчков. Например, после первого щелчка рисуется точка, соответствующая первому щелчку. После второго щелчка – две точки, соответствующие первому и второму щелчкам. После третьего щелчка – три точки, соответствующие первому, второму и третьему щелчкам, и так далее.

Но, вернемся к нашей программе. В строке 62 выполняется очистка элемента `<canvas>`. После этого инструкция `for` (строка 65) последовательно переписывает координаты из массива `g_points` в переменную-атрибут `a_Position` вершинного шейдера. Затем `gl.drawArrays()` рисует точку:

```
65  for(var i = 0; i < len; i+=2) {  
66      // Передать координаты щелчка в переменную a_Position  
67      gl.vertexAttrib3f(a_Position, g_points[i], g_points[i+1], 0.0);
```

Так же, как в программе `HelloPoint2.js`, для передачи координат точки в переменную-атрибут `a_Position` используется метод `gl.vertexAttrib3f()`, ссылка на которую передается функции `click()` в четвертом параметре, в строке 51.

Массив `g_points` хранит координаты X и Y указателя мыши в момент щелчков, как показано на рис. 2.28. То есть, если `g_points[i]` хранит координату X, то `g_points[i+1]` хранит координату Y, поэтому переменная `i` цикла `for` увеличивается в строке 65 на 2.

Теперь, когда все готово к рисованию точки, осталось только нарисовать ее, что и делается вызовом `gl.drawArrays()`:

```
69      // Нарисовать точку  
70      gl.drawArrays(gl.POINTS, 0, 1);
```

Хотя пример получился немного сложным, нетрудно заметить, что применение обработчика событий в сочетании с переменной-атрибутом обеспечивает большую гибкость и универсальность в отображении графики, в зависимости от действий пользователя.

Эксперименты с примером программы

Теперь немного поэкспериментируем с программой `ClickedPoints`. После загрузки файла `ClickedPoints.html` в браузер, в ответ на каждый щелчок в области рисования будет выводиться точка, как показано на рис. 2.26.

Давайте посмотрим, что получится, если не выполнять очистку области рисования перед выводом точек в строке 62. Закомментируйте эту строку, как показано ниже, и перезагрузите страницу в браузере.

```
61      // Очистить <canvas>  
62      // gl.clear(gl.COLOR_BUFFER_BIT);
```

```
63
64   var len = g_points.length;
65   for(var i = 0; i < len; i+=2) {
66       // Передать координаты щелчка в переменную a_Position
67       gl.vertexAttrib3f(a_Position, g_points[i], g_points[i+1], 0.0);
68
69       // Нарисовать точку
70       gl.drawArrays(gl.POINTS, 0, 1);
71   }
72 }
```

Сразу после запуска программы вы увидите черное поле, но после первого же щелчка область рисования окрасится в белый цвет и в ней появится красная точка. Это объясняется тем, что после операции рисования WebGL повторно инициализирует буфер цвета значением по умолчанию (0.0, 0.0, 0.0, 0.0) (см. табл. 2.1). То есть, альфа-канал по умолчанию получает значение 0.0, из-за чего цвет области рисования `<canvas>` становится прозрачным и за ней становится виден фон веб-страницы (белый, в данном случае). Если такое поведение по умолчанию нежелательно, следует установить цвет фона по умолчанию с помощью `gl.clearColor()` и затем всегда вызывать `gl.clear()` перед рисованием.

Интересно также посмотреть, можно ли упростить код. В `ClickedPoints.js` координаты X и Y хранятся в массиве `g_points` по отдельности. Однако есть возможность хранить их вместе, в виде массива:

```
58 // Сохранить координаты в массиве g_points
59 g_points.push([x, y]);
```

В этом случае в каждом элементе массива `g_points` будет сохраняться новый, двухэлементный массив `[x, y]`. Очень удобно, что JavaScript позволяет хранить массивы в массивах.

Извлечь отдельные координаты из массива можно следующим образом: сначала из массива извлекается элемент с парой координат (строка 66), а так как сам элемент так же является массивом с парой координат (X, Y), чтобы получить их, достаточно извлечь первый и второй элементы из этого массива (строка 67):

```
65   for(var i = 0; i < len; i++) {
66       var xy = g_points[i];
67       gl.vertexAttrib3f(a_Position, xy[0], xy[1], 0.0);
68       ...
71   }
```

Благодаря этому приему можно хранить координаты X и Y вместе, что упростит программу и повысит ее читаемость.

Изменение цвета точки

К настоящему моменту у вас должно сложиться достаточно полное представление о работе шейдеров и передаче данных в них из программ на JavaScript. Теперь

мы предлагаем закрепить полученные знания, написав более сложную программу, которая рисует точки цветом, зависящим от их координат в элементе `<canvas>`.

Как изменить цвет точки, мы узнали, когда рассматривали пример `HelloPoint1`. Там мы изменяли сам фрагментный шейдер, подставляя выбранное значение цвета. В этом разделе мы напишем программу, которая будет задавать цвет каждой точки из программного кода на JavaScript. Это чем-то напоминает пример `HelloPoint2`, где мы передавали координаты точки из программы JavaScript в вершинный шейдер. Разница лишь в том, что в этом примере нам нужно будет передавать данные во фрагментный шейдер, а не в вершинный.

Новая программа называется `ColoredPoints`. Если загрузить этот пример в браузер, можно увидеть, что он действует в точности как `ClickedPoints`, за исключением того, что цвет точки зависит от ее координат (см. рис. 2.29).

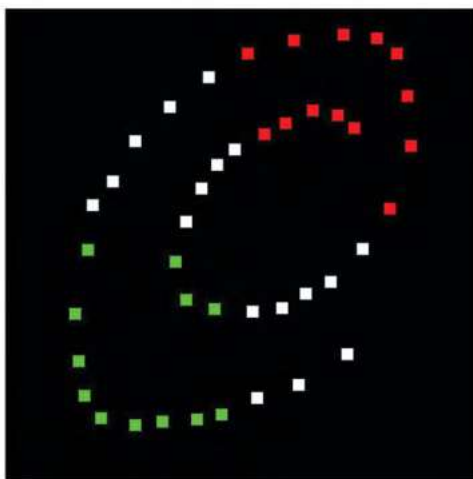


Рис. 2.29. `ColoredPoints`

Для передачи данных во фрагментный шейдер можно использовать `uniform`-переменные (объявленные со спецификатором `uniform`) и реализова те же шаги, что и при использовании переменных-атрибутов, только на этот раз целью является фрагментный шейдер, а не вершинный:

1. объявить `uniform`-переменную во фрагментном шейдере;
2. присвоить встроенной переменной `gl_FragColor` значение `uniform`-переменной;
3. присвоить данные `uniform`-переменной.

Обратимся к примеру программы и посмотрим, как реализуются эти шаги.

Пример программы (`ColoredPoints.js`)

Вершинный шейдер в этой программе остался тем же, что и в программе `ClickedPoints.js`. А вот фрагментный шейдер на этот раз играет более важную роль, потому что программа определяет цвет точки динамически, а цвет, как вы

наверняка помните, обрабатывается фрагментным шейдером. В листинге 2.8 приводится содержимое файла `ColoredPoints.js`.

Листинг 2.8. `ColoredPoints.js`

```

1 // ColoredPoints.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'void main() {\n' +
6   '   gl_Position = a_Position;\n' +
7   '   gl_PointSize = 10.0;\n' +
8   '}\n';
9
10 // Фрагментный шейдер
11 var FSHADER_SOURCE =
12   'precision mediump float;\n' +
13   'uniform vec4 u_FragColor;\n' + // uniform-переменная      <- (1)
14   'void main() {\n' +
15   '   gl_FragColor = u_FragColor;\n' + //                    <- (2)
16   '}\n';
17
18 function main() {
19   ...
20   // Инициализировать шейдеры
21   if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
22     ...
23   }
24
25   // Получить ссылку на переменную-атрибут a_Position
26   var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
27   ...
28   // Получить ссылку на uniform-переменную u_FragColor
29   var u_FragColor = gl.getUniformLocation(gl.program, 'u_FragColor');
30   ...
31   // Зарегистрировать функцию (обработчик) для вызова по щелчку мышью
32   canvas.onmousedown = function(ev){ click(ev, gl, canvas, a_Position,
33                                         ➡u_FragColor) };
34   ...
35   gl.clear(gl.COLOR_BUFFER_BIT);
36 }
37
38
39 var g_points = []; // Массив с координатами точек, где выполнялись щелчки
40 var g_colors = []; // Массив со значениями цветов точек
41 function click(ev, gl, canvas, a_Position, u_FragColor) {
42   var x = ev.clientX; // координата X указателя мыши
43   var y = ev.clientY; // координата Y указателя мыши
44   var rect = ev.target.getBoundingClientRect();
45
46   x = ((x - rect.left) - canvas.width/2)/(canvas.width/2);
47   y = (canvas.height/2 - (y - rect.top))/(canvas.height/2);
48
49   // Сохранить координаты в массиве g_points
50   g_points.push([x, y]);

```

```

71 // Сохранить цвет в массиве g_colors
72 if(x >= 0.0 && y >= 0.0) { // Первый квадрант
73   g_colors.push([1.0, 0.0, 0.0, 1.0]); // Красный
74 } else if(x < 0.0 && y < 0.0) { // Третий квадрант
75   g_colors.push([0.0, 1.0, 0.0, 1.0]); // Зеленый
76 } else { // Остальные
77   g_colors.push([1.0, 1.0, 1.0, 1.0]); // Белый
78 }
79
80 // Очистить <canvas>
81 gl.clear(gl.COLOR_BUFFER_BIT);
82
83 var len = g_points.length;
84 for(var i = 0; i < len; i++) {
85   var xy = g_points[i];
86   var rgba = g_colors[i];
87
88   // Передать координаты щелчка в переменную a_Position
89   gl.vertexAttrib3f(a_Position, xy[0], xy[1], 0.0);
90   // Передать цвет точки в переменную u_FragColor
91   gl.uniform4f(u_FragColor, rgba[0], rgba[1], rgba[2], rgba[3]);    <-(3)
92   // Нарисовать точку
93   gl.drawArrays(gl.POINTS, 0, 1);
94 }
95 }

```

Uniform-переменные

Как вы наверняка помните, переменные-атрибуты используются для передачи данных из программного кода на JavaScript в вершинные шейдеры. К сожалению, переменные-атрибуты доступны только вершинным шейдерам, а во фрагментных шейдерах следует использовать uniform-переменные. Существует также альтернативный механизм – varying-переменные (см. рис. 2.30, внизу) – однако он гораздо сложнее, поэтому мы отложим знакомство с ним до главы 5.



Рис. 2.30. Два способа передачи данных во фрагментный шейдер

Когда мы познакомились с переменными атрибутами, мы видели, что *uniform*-переменные служат для передачи «одинаковых» («*uniform*»), не изменяющихся данных во все вершины или фрагменты в шейдере. Давайте теперь воспользуемся этим свойством.

Прежде чем задействовать *uniform*-переменную, ее необходимо объявить. Объявление должно следовать стандартному шаблону: Спецификатор класса хранения Тип Имя переменной (рис. 2.31), как это делалось, когда мы объявляли переменную-атрибут. (См. раздел «Пример программы (HelloPoint2.js)».)⁴

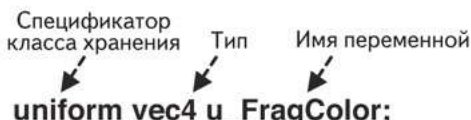
Спецификатор
класса хранения Тип Имя переменной

uniform vec4 u_FragColor;

Рис. 2.31. Объявление *uniform*-переменной

В этом примере программы *uniform*-переменная *u_FragColor* получила свое имя по переменной *gl_FragColor*, потому что значение этой *uniform*-переменной будет присваиваться переменной *gl_FragColor*. Префикс *u_* в имени *u_FragColor* является частью наших соглашений об именовании и указывает, что данная переменная является *uniform*-переменной. Тип переменной *u_FragColor* должен соответствовать типу переменной *gl_FragColor*. Поэтому переменная *u_FragColor* объявляется именно так, как показано в строке 13:

```

10 // Фрагментный шейдер
11 var FSHADER_SOURCE =
12   'precision mediump float;\n' +
13   'uniform vec4 u_FragColor;\n' + // uniform-переменная
14   'void main() {\n' +
15   '   gl_FragColor = u_FragColor;\n' +
16   '}\n';

```

Обратите внимание, что в строке 12 с помощью **спецификатора точности** (*precision*) определяется диапазон и точность представления значений переменными – в данном случае выбрана точность *medium* (средняя) – о котором будет рассказываться в главе 5.

В строке 15 выполняется присваивание значения *uniform*-переменной *u_FragColor* переменной *gl_FragColor*, что вызывает окрашивание рисуемой точки в переданный цвет. Передача цвета через *uniform*-переменную напоминает использование переменных-атрибутов – нужно получить ссылку на переменную и затем в программе JavaScript записывать данные по этой ссылке.

Получение ссылки на *uniform*-переменную

Получить ссылку на *uniform*-переменную можно с помощью следующего метода.

⁴ В языке GLSL ES переменную-атрибут можно объявить только с типом *float*; зато *uniform*-переменные могут быть любого типа. (Подробности см. в главе 6.)

gl.getUniformLocation(program, name)

Возвращает ссылку на *uniform*-переменную, определяемую параметром *name*.

Параметры	<i>program</i>	Объект программы, хранящий вершинный и фрагментный шейдеры.
	<i>name</i>	Определяет имя <i>uniform</i> -переменной, ссылку на которую требуется получить.
Возвращаемое значение	непустое значение	Ссылка на указанную <i>uniform</i> -переменную.
	<i>null</i>	Указанная <i>uniform</i> -переменная не найдена или ее имя начинается с зарезервированного префикса <i>gl_</i> или <i>webgl_</i> .
Ошибки	<i>INVALID_OPERATION</i>	Объект программы <i>program</i> не был скомпонован (см. главу 9).
	<i>INVALID_VALUE</i>	Длина имени <i>name</i> больше максимально возможной длины (256 символов по умолчанию).

Назначение и параметры этого метода полностью совпадают с назначением и параметрами метода `gl.getAttribLocation()`. Но, если запрошенная *uniform*-переменная не существует или имя начинается с зарезервированного префикса, этот метод возвращает *null*, а не `-1`. По этой причине возвращаемое значение следует проверять, сравнивая его со значением *null*. Эту проверку можно видеть в строке 44. В логическом контексте значение *null* в языке JavaScript интерпретируется как `false`, поэтому для проверки результата можно использовать оператор `!`, как показано ниже:

```

42 // Получить ссылку на uniform-переменную u_FragColor
43 var u_FragColor = gl.getUniformLocation(gl.program, 'u_FragColor');
44 if (!u_FragColor) {
45     console.log('Failed to get u_FragColor variable');
46     return;
47 }
```

Присваивание значения *uniform*-переменной

После получения ссылки на *uniform*-переменную, ей можно присвоить значение с помощью метода `gl.uniform4f()`. Эта операция выполняется в строке 41, с помощью метода `gl.uniform4f()`. Он имеет то же назначение и параметры, что и методы `gl.vertexAttrib[1234]f()`.

gl.uniform4f(location, v0, v1, v2, v3)

Присваивает данные (*v0*, *v1*, *v2* и *v3*) *uniform*-переменной, определяемой аргументом *location*.

Параметры	<i>location</i>	Ссылка на <i>uniform</i> -переменную, которой требуется присвоить указанное значение.
------------------	-----------------	---

v0	Значение, используемое как первый элемент для uniform-переменной.
v1	Значение, используемое как второй элемент для uniform-переменной.
v2	Значение, используемое как третий элемент для uniform-переменной.
v3	Значение, используемое как четвертый элемент для uniform-переменной.
Возвращаемое значение	нет
Ошибки	INVALID_OPERATION Объект программы program не был скомпилирован. Значение location является недопустимой ссылкой.

Рассмотрим фрагмент программы, где метод `gl.uniform4f()` применяется для присваивания данных (строка 91). Как можно видеть, вызову этого метода предшествует множество подготовительных операций:

```

71 // Сохранить цвет в массиве g_colors
72 if(x >= 0.0 && y >= 0.0) {           // Первый квадрант
73     g_colors.push([1.0, 0.0, 0.0, 1.0]); // Красный
74 } else if(x < 0.0 && y < 0.0) {       // Третий квадрант
75     g_colors.push([0.0, 1.0, 0.0, 1.0]); // Зеленый
76 } else {                             // Остальные
77     g_colors.push([1.0, 1.0, 1.0, 1.0]); // Белый
78 }
...
83 var len = g_points.length;
84 for(var i = 0; i < len; i++) {
85     var xy = g_points[i];
86     var rgba = g_colors[i];
    ...
91     gl.uniform4f(u_FragColor, rgba[0], rgba[1], rgba[2], rgba[3]);

```

Чтобы понять логику работы программы, давайте вспомним ее цель – рисовать точки разного цвета, в зависимости от их местоположения в области рисования `<canvas>`. Мы решили, что в первом квадранте точки должны быть нарисованы красным цветом; в третьем квадранте – зеленым; в остальных квадрантах – белым (см. рис. 2.32).

Код в строках с 72 по 78 просто определяет, в каком квадранте выполнен щелчок и сохраняет соответствующий цвет в массив `g_colors`. В заключение, в строке 84, программа выполняет обход точек в цикле и записывает в uniform-переменную



Рис. 2.32. Нумерация квадрантов в системе координат и соответствующие им цвета

`u_FragColor` соответствующее значение цвета (строка 91). В процессе обхода все точки сохраняются в буфере цвета, который затем автоматически выводится браузером на экран.

Прежде чем завершить главу, давайте познакомимся с еще одним семейством методов `gl.uniform[1234]f()`.

Семейство методов `gl.uniform4f()`

Метод `gl.uniform4f()` является частью семейства методов. Метод `gl.uniform1f()` устанавливает единственное значение (`v0`), метод `gl.uniform2f()` – два значения (`v0` и `v1`) и метод `gl.uniform3f()` – три значения (`v0`, `v1` и `v2`).

```
gl.uniform1f(location, v0)
gl.uniform2f(location, v0, v1)
gl.uniform3f(location, v0, v1, v2)
gl.uniform4f(location, v0, v1, v2, v3)
```

Присваивают значение переменной-атрибуту по ссылке `location`. Метод `gl.uniform1f()` присваивает единственное значение и используется для изменения первого элемента вектора `uniform`-переменной. Второму и третьему элементам вектора будут присвоены значения 0.0, а четвертому – значение 1.0. Аналогично действует метод `gl.uniform2f()`, который присваивает значения первым двум элементам, третьему присваивает значение 0.0, а четвертому – значение 1.0. Метод `gl.uniform3f()` присваивает значения первым трем элементам, а четвертому – значение 1.0. Метод `gl.uniform4f()` присваивает значения всем четырем элементам вектора.

Параметры	<code>location</code>	Ссылка на переменную-атрибут, которой требуется присвоить указанное значение.
	<code>v0, v1, v2 и v3</code>	Значения, которые должны быть присвоены первому, второму, третьему и четвертому элементам переменной-атрибута.
Возвращаемое значение	нет	
Ошибки	<code>INVALID_OPERATION</code>	Объект программы <code>program</code> не был скомпонован. Значение <code>location</code> является недопустимой ссылкой.

В заключение

В этой главе вы познакомились с основными функциями WebGL и приемами их использования. В частности, вы познакомились с шейдерами – основным механизмом, используемым в WebGL для рисования графики. Опираясь на эти знания, мы написали несколько примеров. Сначала мы создали простую программу, рисующую красную точку, а затем постепенно усложняли ее, добавив возможность рисования точек в позиции указателя мыши во время щелчка и изменение цвета. Все эти примеры должны были помочь вам понять, как передавать данные из программ на JavaScript в шейдеры, что очень важно для понимания будущих примеров.

В этой главе мы ограничились рисованием 2-мерных точек. Однако, полученные здесь знания можно с успехом использовать также для рисования более сложных фигур и трехмерных объектов.

Из всего, о чем рассказывалось в этой главе, важно также запомнить, что вершинный шейдер выполняет операции с вершинами, а фрагментный шейдер – с фрагментами. В следующей главе мы познакомимся с некоторыми другими функциями WebGL, и при этом постепенно будем усложнять фигуры, отображаемые на экране.



ГЛАВА 3.

Рисование и преобразование треугольников

В главе 2, «Первые шаги в WebGL», были представлены основные приемы рисования графики в WebGL. Вы узнали, как получить контекст отображения для WebGL и как очистить `<canvas>` перед выводом в него двух- или трехмерной графики. Затем исследовали назначение и особенности вершинных и фрагментных шейдеров, и как с их помощью осуществляется рисование графики. Затем, опираясь на полученные знания, мы сконструировали несколько простых программ, рисующих точки на экране.

В этой главе, опираясь на основы, заложенные в предыдущей главе, мы посмотрим, как рисовать более сложные фигуры и как манипулировать ими в трехмерном пространстве. В частности, в этой главе рассматриваются:

- критически важная роль треугольников в трехмерной графике и поддержка рисования треугольников в WebGL;
- использование множества треугольников для рисования других фигур;
- основные преобразования: перемещение, вращение и масштабирование с использованием простых уравнений;
- применение матричных операций для упрощения преобразований.

К концу этой главы вы будете иметь исчерпывающее представление о поддержке рисования простых фигур в WebGL и о том, как использовать матричные операции для манипулирования этими фигурами. Эти знания нам пригодятся в главе 4, «Дополнительные преобразования и простая анимация», когда мы займемся исследованием простых анимационных эффектов.

Рисование множества точек

Как вы уже наверняка знаете, трехмерные модели в действительности состоят из множества простых фрагментов: скромных треугольников. Например, взгляните на модель лягушонка, изображенную на рис. 3.1, справа на этом рисунке видно, что вся модель сконструирована из треугольников (и множества вершин). То есть, несмотря на то, что этот персонаж компьютерной игры имеет сложную форму, она составлена из очень простых строительных блоков – треугольников. Используя

все более и более мелкие треугольники (то есть, все большее и большее число вершин) можно создавать более сложные и гладкие объекты. Обычно сложные фигуры и персонажи компьютерных игр состоят из десятков и сотен треугольников. Таким образом, определение множества вершин, составляющих треугольники, является основой рисования трехмерных объектов.

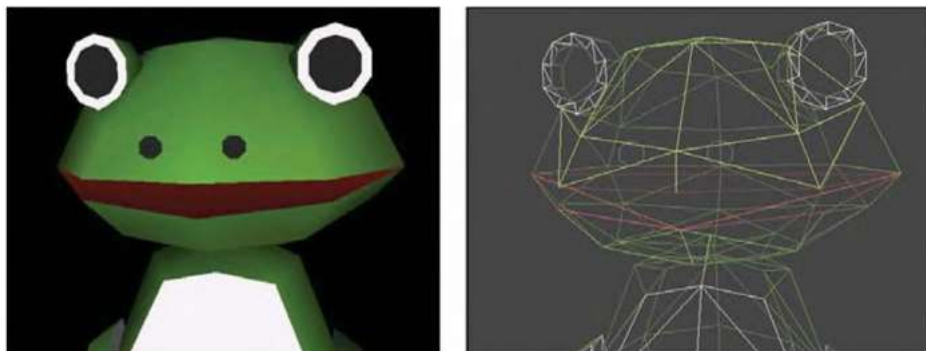


Рис. 3.1. Сложные фигуры конструируются из множества треугольников

В этом разделе мы исследуем процесс рисования фигур с использованием множества вершин. Однако, чтобы избежать ненужных сложностей, мы продолжим использовать двухмерные фигуры, потому что обработка двухмерных фигур с несколькими вершинами ничем не отличается от обработки трехмерных объектов. Фактически, если вы овладеете этими приемами для двухмерных фигур, вы с легкостью разберетесь в остальных примерах в этой книге, где те же приемы применяются для рисования трехмерных объектов.

В качестве примера одновременной работы с несколькими вершинами мы напишем программу `MultiPoint`, которая рисует три красные точки – не забывайте, что три точки, или вершины, образуют треугольник. На рис. 3.2 показано, как выглядит экран программы `MultiPoint`.

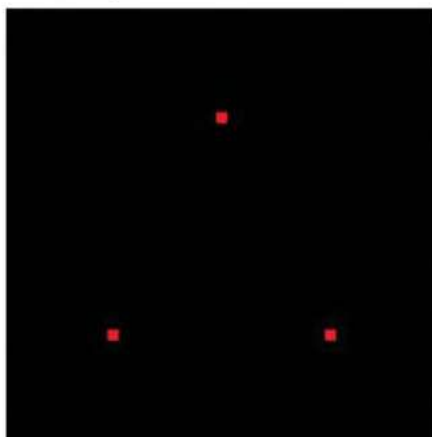


Рис. 3.2. `MultiPoint`

В предыдущей главе мы написали пример программы `ClickedPoints`, которая рисует множество точек, соответствующих щелчкам мышью. `ClickedPoints` хранит координаты щелчков в массиве JavaScript (`g_points[]`) и рисует каждую точку вызовом метода `gl.drawArrays()` (листинг 3.1). Чтобы нарисовать несколько точек, в ней использовался цикл, выполнявший обход массива и рисовавший точки по очереди, передавая в шейдер каждую вершину по отдельности.

Листинг 3.1. Рисование нескольких точек, как это реализовано в `ClickedPoints.js` (глава 2)

```
65 for(var i = 0; i < len; i+=2) {
66     // Передать координаты щелчка в переменную a_Position
67     gl.vertexAttrib3f(a_Position, g_points[i], g_points[i+1], 0.0);
68
69     // Нарисовать точку
70     gl.drawArrays(gl.POINTS, 0, 1);
71 }
```

Очевидно, что это решение можно использовать, только для рисования не связанных между собой точек. Для создания фигур, состоящих из нескольких вершин, таких как треугольники, прямоугольники и кубы, необходим способ одновременной передачи множества вершин в вершинный шейдер.

Система WebGL предусматривает удобный способ передачи множества вершин с использованием, так называемого **буферного объекта**. Буферный объект – это область памяти в системе WebGL, где можно хранить множество вершин. Она используется и как область для сборки фигуры, и как средство передачи сразу нескольких вершин в вершинный шейдер.

Но, перед погружением в исследование буферного объекта, рассмотрим пример программы, чтобы вы представляли, какие операции и в каком порядке выполняются.

Пример программы (*MultiPoint.js*)

Диаграмма с последовательностью операций для `MultiPoint.js` (см. рис. 3.3) близко напоминает аналогичную диаграмму для `ClickedPoints.js` (листинг 2.7) и `ColoredPoints.js` (листинг 2.8), которую мы видели в главе 2. Единственное отличие – новый шаг, связанный с определением координат вершин.

Этот шаг реализован в строке 34, в листинге 3.2, в виде вызова функции `initVertexBuffers()`.

Листинг 3.2. `MultiPoint.js`

```
1 // MultiPoint.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4     'attribute vec4 a_Position;\n' +
5     'void main() {\n' +
6     '    gl_Position = a_Position;\n' +
7     '    gl_PointSize = 10.0;\n' +
```

```
8   '}\n';
9
10  // Фрагментный шейдер
11  ...
12
13  function main() {
14      ...
15      // Получить контекст отображения для WebGL
16      var gl = getWebGLContext(canvas);
17      ...
18      // Инициализировать шейдеры
19      if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
20          ...
21      }
22
23      // Определить координаты вершин
24      var n = initVertexBuffers(gl);
25      if (n < 0) {
26          console.log('Failed to set the positions of the vertices');
27          return;
28      }
29
30      // Указать цвет для очистки области рисования <canvas>
31      ...
32      // Очистить <canvas>
33      ...
34      // Нарисовать три точки
35      gl.drawArrays(gl.POINTS, 0, n); // n равно 3
36  }
37
38  function initVertexBuffers(gl) {
39      var vertices = new Float32Array([
40          0.0, 0.5, -0.5, -0.5, 0.5, -0.5
41      ]);
42      var n = 3; // Число вершин
43
44      // Создать буферный объект
45      var vertexBuffer = gl.createBuffer();
46      if (!vertexBuffer) {
47          console.log('Failed to create the buffer object ');
48          return -1;
49      }
50
51      // Определить тип буферного объекта
52      gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
53      // Записать данные в буферный объект
54      gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
55
56      var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
57      ...
58      // Сохранить ссылку на буферный объект в переменной a_Position
59      gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);
60
61      // Разрешить присваивание переменной a_Position
```

```

77  gl.enableVertexAttribArray(a_Position);
78
79  return n;
80 }

```



Рис. 3.3. Последовательность операций, выполняемых программой MultiPoints.js

Новая функция `initVertexBuffers()` определена в строке 50 и вызывается в строке 34 для подготовки буферного объекта с вершинами. Функция сохраняет множество вершин в буферном объекте и затем передает его вершинному шейдеру:

```

33 // Определить координаты вершин
34 var n = initVertexBuffers(gl);

```

Возвращаемое значение функции – число вершин, которые нужно нарисовать, – сохраняется в переменной `n`. Обратите внимание, что в случае ошибки `n` получает отрицательное значение.

Как и в предыдущих примерах, операция рисования выполняется единственным вызовом `gl.drawArrays()` в строке 48. Этот вызов `gl.drawArrays()` напоминает вызов в примере `ClickedPoints.js`, за исключением того, что в третьем аргументе передается значение `n` вместо 1:

```

46 // Нарисовать три точки
47 gl.drawArrays(gl.POINTS, 0, n); // n равно 3

```

Поскольку передача вершин в вершинный шейдер производится с помощью буферного объекта (внутри функции `initVertexBuffers()`), необходимо передать

функции `gl.drawArrays()` число вершин в третьем параметре, чтобы система WebGL знала, как нарисовать фигуру, используя все вершины в буферном объекте.

Использование буферных объектов

Как отмечалось выше, буферный объект, предоставляемый системой WebGL, является областью памяти (см. рис. 3.4), где хранятся вершины, которые требуется нарисовать. Применение буферного объекта позволяет передать в вершинный шейдер сразу несколько вершин, используя при этом единственную переменную-атрибут.

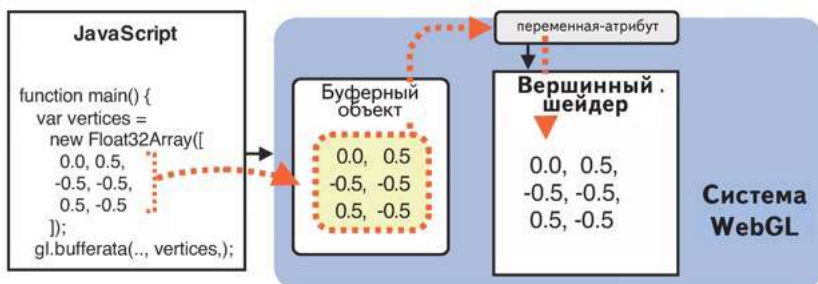


Рис. 3.4. Передача в вершинный шейдер сразу нескольких вершин с помощью буферного объекта

Данные (координаты вершин), что записываются в буферный объект, в программе определяются как специальный массив типа `Float32Array` (см. ниже). Мы обсудим этот тип массивов во всех подробностях позднее, а пока рассматривайте его как самый обычный массив:

```

51 var vertices = new Float32Array([
52     0.0, 0.5, -0.5, -0.5, 0.5, -0.5
53 ]);
  
```

Чтобы посредством буферного объекта передать в вершинный шейдер массив данных, необходимо выполнить пять шагов. Так как аналогичный подход используется системой WebGL для работы с другими объектами, такими как текстурные объекты (глава 4) и объекты кадровых буферов (глава 8, «Освещение объектов»), исследуем эти шаги во всех подробностях, чтобы позднее вы могли опираться на полученные знания:

1. Создать буферный объект (`gl.createBuffer()`).
2. Указать тип буферного объекта (`gl.bindBuffer()`).
3. Записать данные в буферный объект (`gl.bufferData()`).
4. Сохранить ссылку на буферный объект в переменной-атрибуте (`gl.vertexAttribPointer()`).
5. Разрешить присваивание (`gl.enableVertexAttribArray()`).

Рис. 3.5 иллюстрирует выполнение этих пяти шагов.

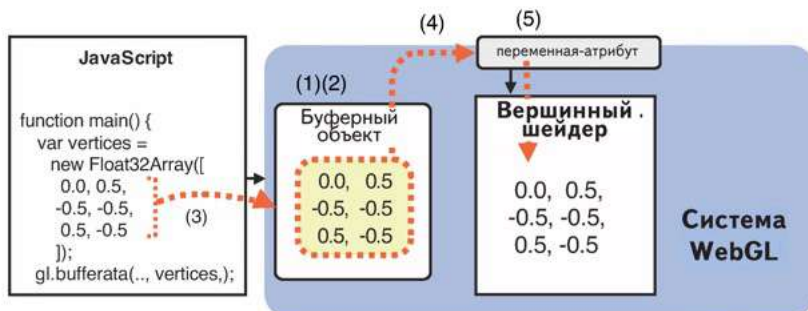


Рис. 3.5. Пять шагов, которые необходимо выполнить для передачи массива данных в вершинный шейдер посредством буферного объекта

В листинге 3.2 эти шаги реализует следующий код:

```

56 // Создать буферный объект <--(1)
57 var vertexBuffer = gl.createBuffer();
58 if (!vertexBuffer) {
59     console.log('Failed to create the buffer object ');
60     return -1;
61 }
62
63 // Указать тип буферного объекта <--(2)
64 gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
65 // Записать данные в буферный объект <--(3)
66 gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
67
68 var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
69 ...
73 // Сохранить ссылку на буферный объект в переменной a_Position <--(4)
74 gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);
75
76 // Разрешить присваивание переменной a_Position <--(5)
77 gl.enableVertexAttribArray(a_Position);

```

Начнем с трех первых шагов (1–3), от создания буферного объекта до записи данных (в данном примере – координат вершин) в буфер, и опишем все методы, используемые на каждом шаге.

Создание буферного объекта (*gl.createBuffer()*)

Прежде, чем задействовать буферный объект, его нужно создать. Это – первый шаг и выполняется он в строке 57:

```
57 var vertexBuffer = gl.createBuffer();
```

Создается буфер вызовом WebGL-метода `gl.createBuffer()`. На рис. 3.6 показано внутреннее состояние WebGL. В верхней части рисунка показано состояние перед вызовом метода, а в нижней части – после вызова. Как видите, метод создает единственный буферный объект в системе WebGL. Назначение ключевых

слов `gl.ARRAY_BUFFER` и `gl.ELEMENT_ARRAY_BUFFER` на рисунке мы разъясним в следующем разделе, так что пока можете просто игнорировать их.

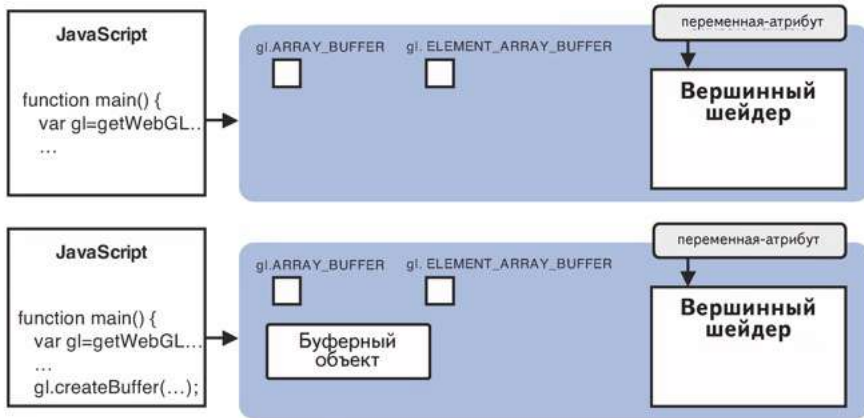


Рис. 3.6. Создание буферного объекта

Ниже приводится описание метода `gl.createBuffer()`.

`gl.createBuffer()`

Создает буферный объект

Возвращаемое значение непустое Ссылка на вновь созданный буферный объект.

`null` Признак ошибки создания буферного объекта.

Ошибки нет

Соответствующий ему метод `gl.deleteBuffer()` удаляет буферный объект, созданный вызовом `gl.createBuffer()`.

`gl.deleteBuffer (buffer)`

Удаляет буферный объект, заданный параметром `buffer`.

Параметры `buffer` Ссылка на буферный объект, подлежащий удалению.

Возвращаемое значение нет

Ошибки нет

Указание типа буферного объекта (`gl.bindBuffer()`)

Второй шаг после создания буферного объекта – указание его «типа». Тип сообщает системе WebGL, какие данные находятся в буферном объекте, что обеспечивает правильную их обработку. Указание типа выполняется в строке 64:

```
64  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
```


Ниже приводится описание метода `gl.bindBuffer()`.

`gl.bindBuffer(target, buffer)`

Активирует буферный объект `buffer` и указывает его тип `target`.

Параметры	Параметр типа <code>target</code> может иметь одно из следующих значений:	
	<code>gl.ARRAY_BUFFER</code>	Указывает, что буферный объект содержит информацию о вершинах.
	<code>gl.ELEMENT_ARRAY_BUFFER</code>	Указывает, что буферный объект содержит значения индексов, ссылающихся на информацию о вершинах. (См. главу 7, «Вперед, в трехмерный мир».)
	<code>buffer</code>	Ссылка на буферный объект, предварительно созданный вызовом <code>gl.createBuffer()</code> . Если в этом параметре передать <code>null</code> , связь с типом <code>target</code> будет разорвана.
Возвращаемое значение	нет	
Ошибки	<code>INVALID_ENUM</code>	В параметре <code>target</code> передано значение, не являющееся ни одной из констант, перечисленных выше. В этом случае продолжит действовать прежняя связь.

В примере программы, рассматриваемом в этом разделе, в параметре `target` передается значение `gl.ARRAY_BUFFER`, так как в буферном объекте хранится информация о вершинах (координаты). После выполнения строки 64 внутреннее состояние системы WebGL изменится, как показано на рис. 3.7.

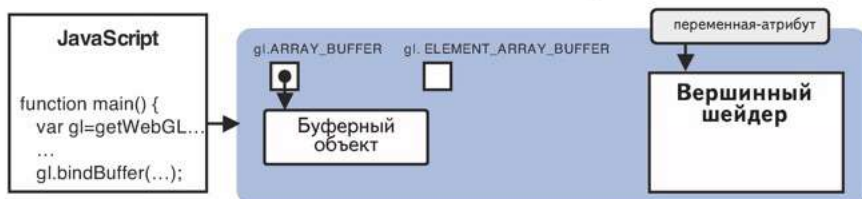


Рис. 3.7. Указание типа буферного объекта

Следующий, третий шаг – запись данных в буферный объект. Обратите внимание: из-за того, что мы не будем пользоваться константой `gl.ELEMENT_ARRAY_BUFFER` до главы 6, она будет убрана из рисунков, следующих ниже.

Запись данных в буферный объект (`gl.bufferData()`)

На третьем шаге выделяется память для буферного объекта и в него записываются данные. Для этого используется метод `gl.bufferData()`, как показано в строке 66:

```
66 gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
```

Этот метод записывает данные, на которые ссылается второй параметр (`vertices`) в буферный объект, тип которого определяется первым параметром (`gl.ARRAY_BUFFER`). После выполнения строки 66, внутреннее состояние системы WebGL изменится, как показано на рис. 3.8.

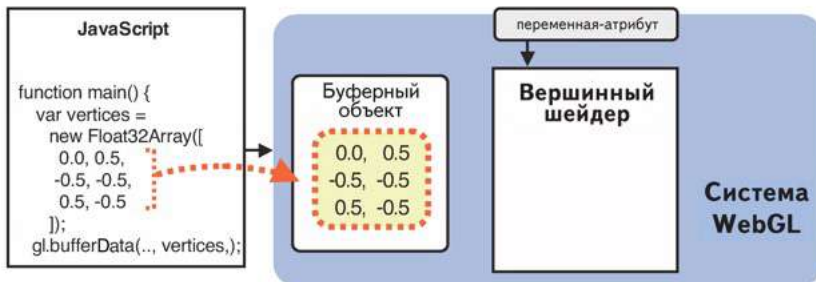


Рис. 3.8. Выделение памяти для буферного объекта и запись данных в него

На этом рисунке видно, что информация о вершинах, определяемая в программе JavaScript, записывается в буферный объект, связанный с `gl.ARRAY_BUFFER`. Ниже приводится описание метода `gl.bufferData()`.

`gl.bufferData(target, data, usage)`

Выделяет память для буферного объекта и записывает данные `data` в буферный объект, тип которого определяется параметром `target`.

Параметры	<code>target</code>	<code>gl.ARRAY_BUFFER</code> или <code>gl.ELEMENT_ARRAY_BUFFER</code>
	<code>data</code>	Данные для записи в буферный объект (типизированный массив; см. следующий раздел).
	<code>usage</code>	Подсказка о том, как программа собирается использовать данные в буферном объекте. Эта подсказка помогает системе WebGL оптимизировать производительность, но не является страховкой от ошибок в вашей программе.
		<code>gl.STATIC_DRAW</code> Данные в буферном объекте будут определены один раз и использованы многократно для рисования фигур. <code>gl.STREAM_DRAW</code> Данные в буферном объекте будут определены один раз и использованы лишь несколько раз для рисования фигур. <code>gl.DYNAMIC_DRAW</code> Данные в буферном объекте будут определены многократно и использованы для рисования фигур так же многократно.

Возвращаемое значение нет

Ошибки `INVALID_ENUM` В параметре `target` передано значение, не являющееся ни одной из констант, перечисленных выше.

Теперь посмотрим, какие данные записываются в буферный объект вызовом `gl.bufferData()`. Для передачи в вершинный шейдер, этому методу передается специальный массив `vertices`, упоминавшийся выше. Сам массив создается в строке 51, с помощью оператора `new` и данными в формате: *<координата X и координата Y первой вершины>*, *<координата X и координата Y второй вершины>*, и так далее:

```
51 var vertices = new Float32Array([
52     0.0, 0.5, -0.5, -0.5, 0.5, -0.5
53 ]);
54 var n = 3; // Число вершин
```

Как видно из предыдущего фрагмента, вместо обычного JavaScript-объекта `Array` используется объект `Float32Array`. Это обусловлено тем, что стандартный массив в JavaScript является универсальной структурой данных, способной хранить числовые данные и строки, и она не оптимизирована для хранения больших объемов данных одного типа, таких как вершины. Чтобы решить эту проблему, в язык были добавлены типизированные массивы, такие как `Float32Array`.

Типизированные массивы

В WebGL часто приходится иметь дело с большими объемами данных одного типа, такими как координаты вершин и цвета, необходимых для рисования трехмерных объектов. С целью оптимизации для каждого типа данных была добавлена поддержка массивов специального типа (**типизированных массивов**). Так как тип данных элементов в типизированных массивах известен заранее, такие массивы могут обрабатываться намного эффективнее.

Массив типа `Float32Array` в строке 51 является примером типизированного массива и обычно используется для хранения координат вершин и цветов. Важно помнить, что WebGL работает с типизированными массивами и такие массивы участвуют во многих операциях, таких как `gl.bufferData()`, в виде второго параметра.

В табл. 3.1 перечислены все поддерживаемые разновидности типизированных массивов. В третьем столбце для справки, сообщаются соответствующие типы данных в языке C для тех, кто знаком с этим языком программирования.

Таблица 3.1. Типизированные массивы, используемые в WebGL

Типизированный массив	Размер одного элемента в байтах	Описание (тип в языке C)
<code>Int8Array</code>	1	8-битовое целое со знаком (<code>signed char</code>)
<code>Uint8Array</code>	1	8-битовое целое без знака (<code>unsigned char</code>)
<code>Int16Array</code>	2	16-битовое целое со знаком (<code>signed short</code>)
<code>Uint16Array</code>	2	16-битовое целое без знака (<code>unsigned short</code>)
<code>Int32Array</code>	4	32-битовое целое со знаком (<code>signed int</code>)
<code>Uint32Array</code>	4	32-битовое целое без знака (<code>unsigned int</code>)
<code>Float32Array</code>	4	32-битовое с плавающей точкой (<code>float</code>)
<code>Float64Array</code>	8	64-битовое с плавающей точкой (<code>double</code>)

Как и обычные массивы JavaScript, типизированные массивы так же имеют собственные методы, свойства и константы, перечисленные в табл. 3.2. Обратите внимание, что в отличие от стандартного объекта `Array`, типизированные массивы не поддерживают методы `push()` и `pop()`.

Таблица 3.2. Методы, свойства, константы типизированных массивов

Методы, свойства, константы	Описание
<code>get(index)</code>	Возвращает элемент с индексом <code>index</code> .
<code>set(index, value)</code>	Записывает в элемент с индексом <code>index</code> значение <code>value</code> .
<code>set(array, offset)</code>	Копирует значения элементов из массива <code>array</code> в данный массив, начиная с элемента с индексом <code>offset</code> .
<code>length</code>	Число элементов в массиве.
<code>BYTES_PER_ELEMENT</code>	Размер одного элемента массива в байтах.

Типизированные массивы, как и стандартные, создаются с помощью оператора `new`, которому передаются исходные данные для массива. Например, чтобы создать массив вершин типа `Float32Array`, можно передать оператору `new` литерал массива `[0.0, 0.5, -0.5, -0.5, 0.5, -0.5]`. Обратите внимание, что оператор `new` является единственным способом создания типизированных массивов. В отличие от объекта `Array`, типизированные массивы не поддерживают оператор `[]`:

```
51 var vertices = new Float32Array([
52     0.0, 0.5, -0.5, -0.5, 0.5, -0.5
53 ]);
```

Кроме того, имеется возможность создать пустой типизированный массив, передав конструктору число элементов в массиве. Например:

```
var vertices = new Float32Array(4);
```

На этом мы заканчиваем исследование первых трех этапов создания и использования буфера (то есть, этапов создания буферного объекта в системе WebGL, указания типа этого объекта и записи данных в него). Давайте теперь посмотрим, как фактически используется буфер – шаги 4 и 5 в описываемом процессе.

Сохранение ссылки на буферный объект в переменной-атрибуте (`gl.vertexAttribPointer()`)

Как рассказывалось в главе 2, присваивание значений переменным-атрибутам можно выполнять с помощью семейства методов `gl.vertexAttrib[1234]f()`. Однако эти методы могут использоваться только для присваивания единственного значения. А как быть, если потребуется присвоить массив значений – вершин, в данном случае?

Для решения этой задачи можно использовать метод `gl.vertexAttribPointer()`, осуществляющий присваивание буферного объекта (в действительности – ссылки, или дескриптора, на буферный объект) переменной-атрибуту. Этот метод при-

меняется в строке 74 для присваивания буферного объекта переменной-атрибуту `a_Position`:

```
74 gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);
```

Описание метода `gl.vertexAttribPointer()` приводится ниже.

`gl.vertexAttribPointer(location, size, type, normalized, stride, offset)`

Присваивает буферный объект типа `gl.ARRAY_BUFFER` переменной-атрибуту `location`.

Параметры	<code>location</code>	Определяет переменную-атрибут, которой будет выполнено присваивание.		
	<code>size</code>	Определяет число компонентов на вершину в буферном объекте (допустимыми являются значения от 1 до 4). Если значение параметра <code>size</code> меньше числа элементов, требуемых переменной-атрибутом, отсутствующие компоненты автоматически получают значения по умолчанию, как указывается в описании семейства методов <code>gl.vertexAttrib[1234]f()</code> . Например, если в параметре передать значение 1, второй и третий компоненты получают значение 0.0, а четвертый элемент – значение 1.0.		
	<code>type</code>	Определяет формат данных. Может иметь одно из следующих значений:		
		<code>gl.UNSIGNED_BYTE</code>	беззнаковый байт	для <code>Uint8Array</code>
		<code>gl.SHORT</code>	короткое целое со знаком	для <code>Int16Array</code>
		<code>gl.UNSIGNED_SHORT</code>	короткое целое без знака	для <code>Uint16Array</code>
		<code>gl.INT</code>	целое со знаком	для <code>Int32Array</code>
		<code>gl.UNSIGNED_INT</code>	целое без знака	для <code>Uint32Array</code>
		<code>gl.FLOAT</code>	вещественное	для <code>Float32Array</code>
	<code>normalized</code>	Либо <code>true</code> , либо <code>false</code> . Указывает на необходимость нормализации невещественных данных в диапазон <code>[0, 1]</code> или <code>[-1, 1]</code> .		
	<code>stride</code>	Определяет число байтов между разными элементами данных. Значение по умолчанию: 0.		
	<code>offset</code>	Определяет смещение (в байтах) от начала буферного объекта, где хранятся данные для вершин. Если данные хранятся, начиная с самого начала буфера, в этом параметре следует передать значение 0.		
Возвращаемое значение	нет			
Ошибки	<code>INVALID_OPERATION</code>	Текущий объект программы не был скомпонован.		
	<code>INVALID_ENUM</code>	Значение параметра <code>location</code> больше или равно максимальному числу переменных-атрибутов (8, по умолчанию). Параметр <code>stride</code> и/или <code>offset</code> имеет отрицательное значение.		

Итак, после выполнения этого четвертого шага, подготовка буферного объекта к использованию практически закончена. Как можно видеть на рис. 3.9, несмотря на то, что буферный объект уже присвоен переменной-атрибуту, WebGL требует выполнить еще один, последний шаг: «разрешить» присваивание и тем самым «замкнуть» цепь.

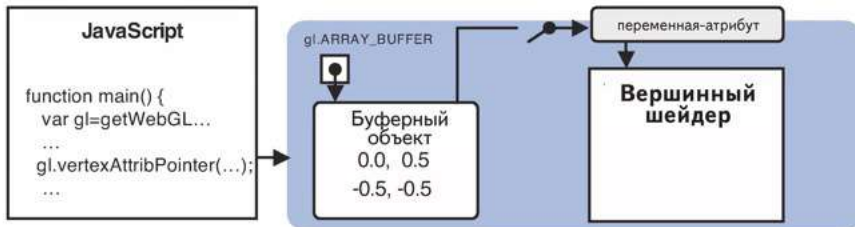


Рис. 3.9. Присваивание буферного объекта переменной атрибуту

Пятый и последний шаг – разрешение присваивания буферного объекта переменной-атрибуту.

Разрешение присваивания переменной-атрибуту (`gl.enableVertexAttribArray()`)

Чтобы сделать буферный объект доступным для вершинного шейдера, необходимо разрешить его присваивание переменной-атрибуту вызовом `gl.enableVertexAttribArray()`, как показано в строке 77:

```
77 gl.enableVertexAttribArray(a_Position);
```

Описание метода `gl.enableVertexAttribArray()` приводится ниже. Обратите внимание, что мы передаем методу буфер, хотя его имя предполагает, что он работает только с массивами вершин (vertex arrays). Мы не допустили никакой ошибки, просто такое имя метода было унаследовано из OpenGL.

`gl.enableVertexAttribArray(location)`

Разрешает присваивание буферного объекта переменной-атрибуту, определяемой параметром `location`.

Параметры	<code>location</code>	Определяет переменную-атрибут.
Возвращаемое значение	нет	
Ошибки	<code>INVALID_VALUE</code>	Значение параметра <code>location</code> больше или равно максимальному числу переменных-атрибутов (8, по умолчанию).

В вызов метода `gl.enableVertexAttribArray()` требуется передать переменную-атрибут, которой прежде был присвоен буферный объект. В результате присваивание разрешается и разорванная цепь замыкается, как показано на рис. 3.10.

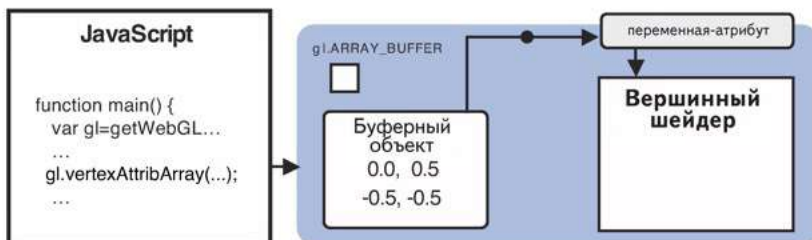


Рис. 3.10. Разрешить присваивание буферного объекта переменной атрибуту

Запретить присваивание (разорвать цепь) можно с помощью метода `gl.disableVertexAttribArray()`.

`gl.disableVertexAttribArray(location)`

Запрещает присваивание буферного объекта переменной-атрибуту, определяемой параметром `location`.

Параметры	<code>location</code>	Определяет переменную-атрибут.
Возвращаемое значение	нет	
Ошибки	<code>INVALID_VALUE</code>	Значение параметра <code>location</code> больше или равно максимальному числу переменных-атрибутов (8, по умолчанию).

Теперь все готово! Осталось лишь выполнить вершинный шейдер, который рисует точки, используя координаты вершин из буферного объекта. Как и в главе 2, эта операция выполняется вызовом `gl.drawArrays`, но, так как выполняется рисование сразу нескольких точек, необходимо задействовать второй и третий параметры этого метода.

Обратите внимание, что после разрешения присваивания нельзя присваивать данные переменной-атрибуту вызовом `gl.vertexAttrib[1234]f()`. Предварительно необходимо явно запретить присваивание буферного объекта – оба метода нельзя использовать одновременно.

Второй и третий параметры метода `gl.drawArrays()`

Прежде чем перейти к подробному описанию этих параметров, давайте познакомимся поближе с методом `gl.drawArrays()`, который был представлен в главе 2. Ниже приводится фрагмент описания метода, включающий только интересующую нас информацию.

gl.drawArrays(mode, first, count)

Выполняет вершинный шейдер, чтобы нарисовать фигуры, определяемые параметром `mode`.

Параметры	mode	Определяет тип фигуры. Допустимыми значениями являются следующие константы: <code>gl.POINTS</code> , <code>gl.LINES</code> , <code>gl.LINE_STRIP</code> , <code>gl.LINE_LOOP</code> , <code>gl.TRIANGLES</code> , <code>gl.TRIANGLE_STRIP</code> и <code>gl.TRIANGLE_FAN</code> .
	first	Определяет номер вершины, с которой должно начинаться рисование (целое число).
	count	Определяет количество вершин (целое число).

В примере программы этот метод используется, как показано ниже:

```
47 gl.drawArrays(gl.POINTS, 0, n); // n равно 3
```

Как и в предыдущих примерах, так как мы рисуем всего лишь три точки, в первом параметре по-прежнему передается значение `gl.POINTS`. Во втором параметре `first` передается 0, потому что информация о вершинах располагается с самого начала буфера. В третьем параметре `count` передается 3, потому что требуется нарисовать три точки (в строке 47, `n` равно 3).

Выполняя строку 47, программа фактически выполняет вершинный шейдер `count` (три) раз, последовательно передавая ему координаты вершин, хранящиеся в буферном объекте, через переменную-атрибут (см. рис. 3.11).

Обратите внимание, что для каждого прогона вершинного шейдера, компонентам `z` и `w` в переменной `a_Position` автоматически присваиваются значения 0.0 и 1.0, потому что эта переменная требует наличия четырех компонентов (`vec4`), а мы передали только два.

Напомним, что в строке 74 второй параметр `size` в вызове метода `gl.vertexAttribPointer()` имеет значение 2. Как уже говорилось выше, второй параметр определяет число координат на вершину в буферном объекте, а так как для каждой вершины мы указали только координаты `X` и `Y`, мы передали в параметре `size` значение 2:

```
74 gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);
```

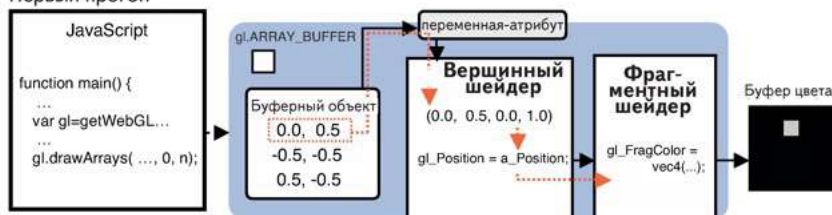
После рисования всех точек, содержимое буфера цвета автоматически отображается в окне браузера (рис. 3.11, внизу), в результате получается картина, изображенная на рис. 3.2.

Эксперименты с примером программы

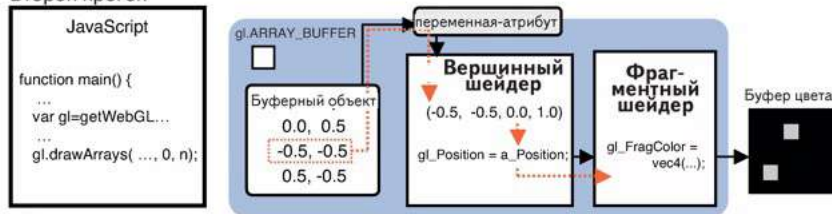
А теперь поэкспериментируем с программой, чтобы лучше понять, как действует `gl.drawArrays()`, и попробуем поиграть со вторым и третьим его параметрами. Сначала передадим 1 в третьем параметре `count`, в строке 47, вместо переменной `n` (имеющей значение 3):

```
47 gl.drawArrays(gl.POINTS, 0, 1);
```

Первый прогон



Второй прогон



Третий прогон



Рис. 3.11. Так данные из буферного объекта передаются в вершинный шейдер

В этом случае вершинный шейдер будет выполнен только один раз и нарисована будет только одна точка – первая вершина в буферном объекте.

Если передать 1 во втором параметре, нарисована будет только вторая точка. Это объясняется тем, что передав 1 во втором параметре, мы сообщили WebGL начать рисование со второй вершины и нарисовать только одну вершину. То есть, на экране снова появится только одна точка, но на этот раз, соответствующая второй вершине:

```
47 gl.drawArrays(gl.POINTS, 1, 1);
```

Этот эксперимент позволяет быстро уловить назначение параметров `first` и `count`. Но что произойдет, если изменить первый параметр `mode`? Об этом подробно рассказывается в следующем разделе.

Привет, треугольник

Теперь, после знакомства с основными приемами передачи вершинному шейдеру координат сразу нескольких вершин, можно попробовать рисовать другие фигуры. В этом разделе демонстрируется пример программы `HelloTriangle`, рисующей единственный 2-мерный треугольник. На рис. 3.12 показан результат работы этой программы.

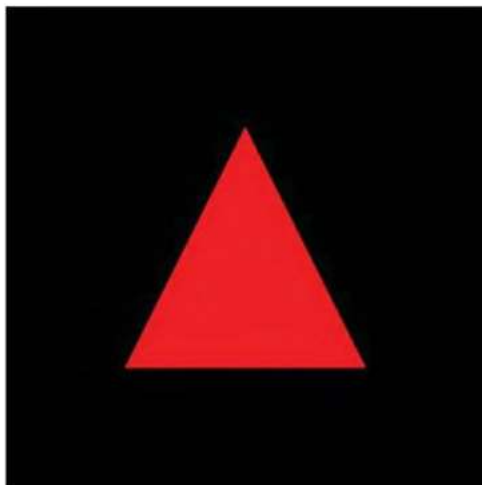


Рис. 3.12. HelloTriangle

Пример программы (HelloTriangle.js)

В листинге 3.3 приводится содержимое файла `HelloTriangle.js`. Оно практически идентично содержимому файла `MultiPoint.js`, рассматривавшегося в предыдущем разделе, за исключением двух важных отличий.

Листинг 3.3. HelloTriangle.js

```
1 // HelloTriangle.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'void main() {\n' +
6   '  gl_Position = a_Position;\n' +
7   '}\n';
8
9 // Фрагментный шейдер
10 var FSHADER_SOURCE =
11   'void main() {\n' +
12   '  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);\n' +
13   '}\n';
14
15 function main() {
16   ...
19   // Получить контекст отображения для WebGL
20   var gl = getWebGLContext(canvas);
21   ...
26   // Инициализировать шейдеры
27   if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
28     ...
29   }
30 }
```

```
31
32 // Определить координаты вершин
33 var n = initVertexBuffers(gl);
...
39 // Указать цвет для очистки области рисования <canvas>
...
45 // Нарисовать треугольник
46 gl.drawArrays(gl.TRIANGLES, 0, n);
47 }
48
49 function initVertexBuffers(gl) {
50   var vertices = new Float32Array([
51     0.0, 0.5, -0.5, -0.5, 0.5, -0.5
52   ]);
53   var n = 3; // Число вершин
...
78   return n;
79 }
```

Вот эти отличия от `MultiPoint.js`:

- из вершинного шейдера была удалена строка `gl_PointSize = 10.0;`, определяющая размер точки; эта строка нужна была только для рисования точек;
- в строке 46 значение `gl.POINTS` первого параметра в вызове метода `gl.drawArrays()` мы заменили значением `gl.TRIANGLES`.

Манипулируя значением первого параметра, `mode`, метода `gl.drawArrays()`, можно рисовать разные фигуры. Давайте посмотрим, как.

Простые фигуры

Изменяя значение первого параметра метода `gl.drawArrays()`, можно изменить прежний смысл строки 46 на «выполнить вершинный шейдер трижды (n равно 3) и нарисовать треугольник по трем вершинам в буфере, начав рисование с первой группы координат»:

```
46 gl.drawArrays(gl.TRIANGLES, 0, n);
```

В данном случае три вершины в буферном объекте больше не являются отдельными точками, а превращаются в три вершины треугольника.

Метод `gl.drawArrays()` обладает большой гибкостью, позволяя указывать семь разных типов фигур с помощью первого аргумента. Эти значения перечислены ниже, в табл. 3.3. Обратите внимание, что значения `v0`, `v1`, `v2` ... соответствуют вершинам в буферном объекте. Порядок следования вершин влияет на рисование фигуры.

В таблице перечислены только фигуры, которые могут быть нарисованы средствами WebGL непосредственно, но из них можно конструировать весьма сложные трехмерные изображения. (Вспомните пример лягушонка, приводившийся в начале этой главы.)

Таблица 3.3. Простые фигуры, поддерживаемые в WebGL

Фигура	Значение параметра mode	Описание
Точка	<code>gl.POINTS</code>	Группа точек. Точки рисуются в координатах вершин $v_0, v_1, v_2 \dots$
Отрезок	<code>gl.LINES</code>	Группа отдельных отрезков. Отрезки рисуются между парами вершин $(v_0, v_1), (v_0, v_1), (v_0, v_1), \dots$. Если число вершин нечетное, последняя вершина игнорируется.
Ломаная	<code>gl.LINE_STRIP</code>	Группа связанных между собой отрезков. Отрезки рисуются между парами вершин $(v_0, v_1), (v_1, v_2), (v_2, v_3), \dots$. Первая вершина становится началом первого отрезка, вторая вершина становится концом первого отрезка и началом второго, и так далее. Вершина с индексом i (где $i > 1$) становится началом i -го отрезка и концом отрезка с индексом $i-1$. (Последняя вершина становится концом последнего отрезка.)
Замкнутая ломаная	<code>gl.LINE_LOOP</code>	Группа связанных между собой отрезков. Вдобавок к обычной ломаной, которая рисуется при значении <code>gl.LINE_STRIP</code> параметра mode, рисуется отрезок, соединяющий последнюю и первую вершины. Отрезки рисуются между парами вершин $(v_0, v_1), (v_1, v_2), \dots$, и (v_n, v_0) , где v_n – последняя вершина.
Треугольник	<code>gl.TRIANGLES</code>	Группа отдельных треугольников. Треугольники задаются триадами вершин $(v_0, v_1, v_2), (v_3, v_4, v_5), \dots$. Если число вершин не кратно 3, лишние вершины игнорируются.
Треугольники с общими сторонами	<code>gl.TRIANGLE_STRIP</code>	Группа треугольников, связанных между собой общими сторонами. Первые три вершины образуют первый треугольник, а второй треугольник образуется из следующей вершины и двух предшествующих, входящих в состав первого треугольника. Треугольники рисуются между тройками вершин $(v_0, v_1, v_2), (v_2, v_1, v_3), (v_2, v_3, v_4) \dots$ (Обратите внимание на порядок следования вершин.)
Треугольники с общей вершиной	<code>gl.TRIANGLE_FAN</code>	Группа треугольников, связанных между собой общими вершинами. Первые три вершины образуют первый треугольник, а второй треугольник образуется из следующей вершины, одной стороны предыдущего треугольника и первой вершины. Треугольники рисуются между тройками вершин $(v_0, v_1, v_2), (v_0, v_2, v_3), (v_0, v_3, v_4), \dots$

Все эти фигуры изображены на рис. 3.13.

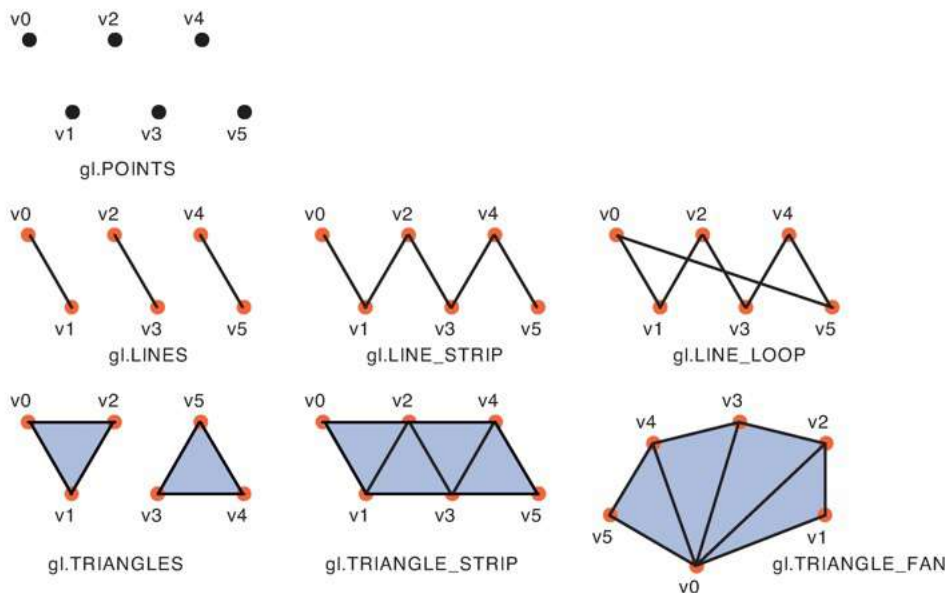


Рис. 3.13. Простые фигуры, поддерживаемые в WebGL

Как показано на рис. 3.13, WebGL может рисовать только три типа фигур: точку, отрезок и треугольник. Однако, как отмечалось в начале главы, сферы, кубы, трехмерные монстры и прочие персонажи компьютерных игр можно конструировать из маленьких треугольников. То есть, с помощью этих простейших фигур можно нарисовать все, что угодно.

Эксперименты с примером программы

Чтобы увидеть, что произойдет, если в первом аргументе передать `gl.LINES`, `gl.LINE_STRIP` и `gl.LINE_LOOP`, внесем изменения в вызов `gl.drawArrays()`, как показано ниже. В результате у нас должно получиться три программы, которым мы предлагаем дать имена `HelloTriangle_LINES`, `HelloTriangle_LINE_STRIP` и `HelloTriangle_LINE_LOOP`, соответственно:

```
46 gl.drawArrays(gl.LINES, 0, n);
46 gl.drawArrays(gl.LINE_STRIP, 0, n);
46 gl.drawArrays(gl.LINE_LOOP, 0, n);
```

На рис. 3.14 показаны результаты выполнения этих трех программ.

Как видите, со значением `gl.LINES` в параметре `mode` метод использует только две первые вершины и игнорирует третью (последнюю), со значением `gl.LINE_STRIP` он рисует два отрезка, используя три первые вершины. Наконец, со значением `gl.LINE_LOOP` он рисует отрезки, как и во втором случае, но затем

«закрывает» получившуюся ломаную, соединяя последнюю и первую вершины, в результате чего получается треугольник.



Рис. 3.14. `gl.LINES`, `gl.LINE_STRIP` и `gl.LINE_LOOP`

Привет, прямоугольник (HelloQuad)

Давайте теперь попробуем таким же способом, каким мы рисовали треугольник, нарисовать прямоугольник. Эта программа называется *HelloQuad*, а на рис. 3.15 показан результат ее работы в браузере.

На рис. 3.16 показаны вершины прямоугольника. Конечно, число вершин равно четырем, потому что это все-таки прямоугольник (то есть, четырехугольник!). Как уже говорилось в предыдущем разделе, WebGL не может рисовать четырехугольники непосредственно, поэтому нужно разбить прямоугольник на два треугольника (v_0, v_1, v_2) и (v_2, v_1, v_3), и затем нарисовать их, передав в параметре `mode` значение `gl.TRIANGLES`, `gl.TRIANGLE_STRIP` или `gl.TRIANGLE_FAN`. В этом примере мы воспользуемся значением `gl.TRIANGLE_STRIP`, потому что только оно позволяет указать всего четыре вершины. Если бы мы решили использовать значение `gl.TRIANGLES`, нам пришлось бы определить шесть вершин.

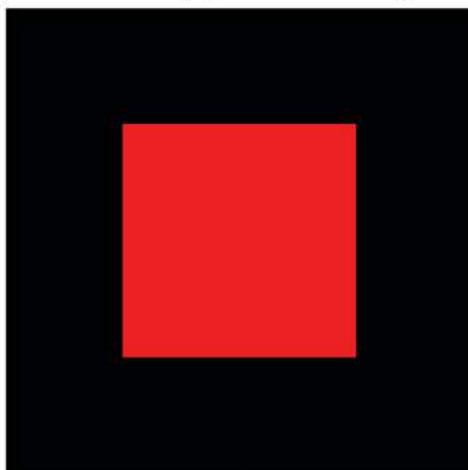


Рис. 3.15. *HelloQuad*

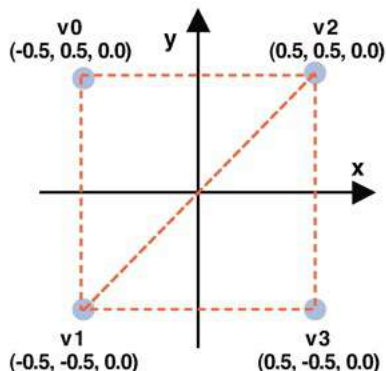


Рис. 3.16. Четыре координаты, необходимые для рисования прямоугольника

Во взятый за основу пример `HelloTriangle.js` нужно добавить координаты дополнительной вершины в строке 50. Обратите внимание, в каком порядке следуют вершины, если он будет иной, команда рисования будет работать неправильно:

```
50  var vertices = new Float32Array([
51    -0.5, 0.5, -0.5, -0.5, 0.5, 0.5, 0.5, -0.5
52  ]);
```

Поскольку теперь у нас четыре вершины, необходимо также изменить число вершин в строке 53 с 3 на 4:

```
53  var n = 4; // Число вершин
```

Затем нужно изменить строку 46, как показано ниже, чтобы программа нарисовала прямоугольник:

```
46  gl.drawArrays( gl.TRIANGLE_STRIP , 0, n);
```

Эксперименты с примером программы

Теперь, когда вы получили представление о том, как использовать значение `gl.TRIANGLE_STRIP`, попробуем заменить его значением `gl.TRIANGLE_FAN`. Соответствующую программу мы назвали `HelloQuad_FAN`:

```
46  gl.drawArrays( gl.TRIANGLE_FAN , 0, n);
```

На рис. 3.17 показан результат работы программы `HelloQuad_FAN`. На этот раз получилась фигура, напоминающая флажок.

Взгляните на порядок следования вершин в программе и расположение треугольников (рис. 3.17, справа), рисуемых методом `gl.drawArrays()`, когда он получает параметр `mode` со значением `gl.TRIANGLE_FAN`. Это объясня-

ет, почему получилась фигура, напоминающая флажок. Фактически, значение `gl.TRIANGLE_FAN` вынуждает WebGL рисовать второй треугольник, первая вершина которого совпадает с первой вершиной первого треугольника (v_0), и наложение второго треугольника на первый создает такой интересный эффект.

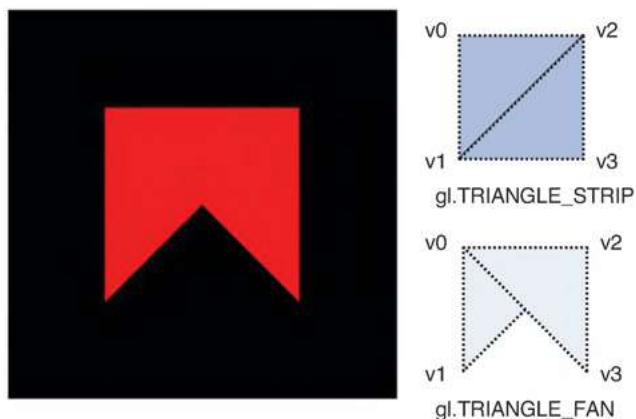


Рис. 3.17. HelloQuad_FAN

Перемещение, вращение и масштабирование

Теперь, после знакомства с основами рисования простых фигур, таких как треугольники и прямоугольники, сделаем еще один шаг вперед и попробуем реализовать перемещение (трансляцию), вращение и масштабирование треугольника, отображение результатов на экране. Эти операции называют **преобразованиями (аффинными преобразованиями)**. В данном разделе будет представлен математический аппарат, описывающий каждую трансформацию, что поможет вам понять, как может быть реализована каждая из операций. Однако, когда вы будете писать собственные программы, этот математический аппарат вам не потребуется – вы просто будете использовать одну из нескольких вспомогательных библиотек, описываемых в следующем разделе и реализующих все необходимые математические вычисления.

Если вы сочтете, что для первого знакомства этот раздел содержит слишком много математической теории, просто пропустите его и вернитесь к нему позже. Или, если вы уже знаете, как реализуются трансформации с применением матриц, вы так же можете пропустить этот раздел.

Итак, для начала напомним программу `TranslatedTriangle`, которая будет перемещать треугольник со стороной 0.5 единицы вправо и вверх на 0.5 единицы. За основу возьмем программу рисования треугольника из предыдущего раздела. Направление «вправо» означает «в сторону положительных значений» по оси X , и направление «вверх» также означает «в сторону положительных значений», но

уже по оси Y. (См. описание системы координат в главе 2.) На рис. 3.18 показан результат работы программы `TranslatedTriangle`.

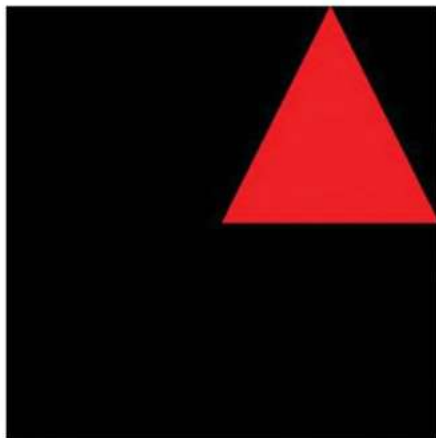


Рис. 3.18. `TranslatedTriangle`

Перемещение

Давайте посмотрим, какие операции требуется применить к координатам каждой вершины фигуры, чтобы выполнить ее перемещение. Фактически нам нужно всего лишь добавить расстояние перемещения по каждому направлению (X и Y) к каждому компоненту координат. Взгляните на рис. 3.19, наша цель – переместить точку $p(x, y, z)$ в точку $p'(x', y', z')$, где расстояние перемещения по осям X, Y и Z равно T_x , T_y и T_z , соответственно. Расстояние T_z на этом рисунке равно 0.

Чтобы определить координаты точки p' , достаточно просто прибавить значения T , как показано в формуле 3.1.

Формула 3.1

$$\begin{aligned}x' &= x + T_x \\y' &= y + T_y \\z' &= z + T_z\end{aligned}$$

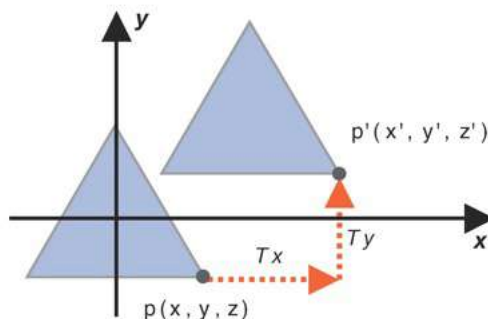


Рис. 3.19. Вычисление расстояний перемещения

Эти простые уравнения можно реализовать в приложении простым прибавлением константы к каждой координате. Возможно вы заметили, что эти операции должны выполняться **для каждой вершины**, поэтому их следует реализовать в вершинном шейдере. Кроме того, совершенно очевидно, что они не имеют никакого отношения к операциям с фрагментами, поэтому оставим фрагментный шейдер в покое.

Поняв суть необходимых вычислений, реализовать их не составит никакого труда. Нам нужно передать расстояния T_x , T_y и T_z в вершинный шейдер, применить формулу 3.1 и присвоить результат переменной `gl_Position`. Давайте рассмотрим программу, реализующую это.

Пример программы (TranslatedTriangle.js)

В листинге 3.4 приводится содержимое файла `TranslatedTriangle.js`, в котором в вершинный шейдер добавлена операция перемещения. Но фрагментный шейдер остался прежним, как в программе `HelloTriangle.js` из предыдущего раздела. Для поддержки изменений в вершинном шейдере, в функцию `main()` был добавлен дополнительный код.

Листинг 3.4. `TranslatedTriangle.js`

```
1 // TranslatedTriangle.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'uniform vec4 u_Translation;\n' +
6   'void main() {\n' +
7     '  gl_Position = a_Position + u_Translation;\n' +
8   '}\n';
9
10 // Фрагментный шейдер
11 ...
12 // Расстояния трансляции по осям X, Y и Z
13 var Tx = 0.5, Ty = 0.5, Tz = 0.0;
14
15 function main() {
16   ...
17   // Получить контекст отображения для WebGL
18   var gl = getWebGLContext(canvas);
19   ...
20   // Инициализировать шейдеры
21   if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
22     ...
23   }
24   // Определить координаты вершин
25   var n = initVertexBuffers(gl);
26   ...
27   // Передать расстояния перемещения в вершинный шейдер
28   var u_Translation = gl.getUniformLocation(gl.program, 'u_Translation');
```

```

...
49 gl.uniform4f(u_Translation, Tx, Ty, Tz, 0.0);
50
51 // Указать цвет для очистки области рисования <canvas>
...
57 // Нарисовать треугольник
58 gl.drawArrays(gl.TRIANGLES, 0, n);
59 }
60
61 function initVertexBuffers(gl) {
62     var vertices = new Float32Array([
63         0.0, 0.0, 0.5, -0.5, -0.5, 0.5, -0.5,
64     ]);
65     var n = 3; // Число вершин
...
90     return n;
93 }

```

Сначала рассмотрим функцию `main()`. В строке 17 определяются переменные для каждого расстояния перемещения из уравнения 3.1:

```
17 var Tx = 0.5, Ty = 0.5, Tz = 0.0;
```

Так как значения T_x , T_y и T_z являются фиксированными (общими для всех вершин), мы использовали `uniform`-переменную¹ `u_Translation` для передачи этих значений в вершинный шейдер. В строке 44 извлекается ссылка на `uniform`-переменную и в строке 49 в нее записываются данные:

```

44 var u_Translation = gl.getUniformLocation(gl.program, 'u_Translation');
...
49 gl.uniform4f(u_Translation, Tx, Ty, Tz, 0.0);

```

Обратите внимание, что метод `gl.uniform4f()` требует передачи однородных координат, поэтому в четвертом аргументе (*w*) мы передаем значение 0.0. Подробнее об этом рассказывается ниже.

А теперь перейдем к вершинному шейдеру, использующему расстояния перемещения. Как видите, `uniform`-переменная `u_Translation`, посредством которой передаются расстояния перемещения, определена в шейдере с типом `vec4` (строка 5). Это обусловлено тем, что нам требуется прибавить значения компонентов `u_Translation` к координатам вершин, которые передаются в переменной `a_Position`, и затем присвоить результат переменной `gl_Position`, также имеющей тип `vec4`. Как уже говорилось в главе 2, операция присваивания в языке GLSL ES допустима только для переменных одного типа:

```

4 'attribute vec4 a_Position;\n' +
5 'uniform vec4 u_Translation;\n' +
6 'void main() {\n' +
7 '     gl_Position = a_Position + u_Translation;\n' +
8 ' }\n';

```

¹ В данном случае спецификатор `uniform` так и переводится на русский язык: «общая для всех». — *Прим. перев.*

После выполнения подготовительных операций, сами вычисления реализуются достаточно просто. Чтобы выполнить вычисления по формуле 3.1 в вершинном шейдере, достаточно просто сложить все расстояния перемещения (T_x , T_y , T_z), переданные в переменной `u_Translation`, с координатами вершины (X , Y , Z) в переменной `a_Position`.

Поскольку обе переменные имеют тип `vec4`, можно использовать обычный оператор `+`, который выполнит попарное сложение всех четырех компонентов (см. рис. 3.20). Эта простая операция сложения векторов является особенностью языка GLSL ES и более подробно будет описываться в главе 6.

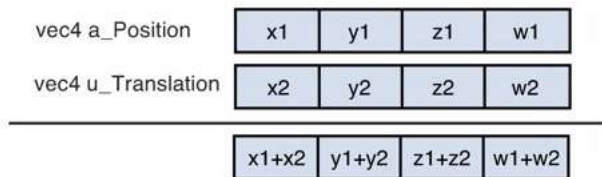


Рис. 3.20. Сложение переменных типа `vec4`

Теперь вернемся к четвертому элементу (w) вектора. Как говорилось в главе 2, в переменной `gl_Position` нужно указать однородные, четырехмерные координаты. Если последний компонент однородных координат равен 1.0, такие координаты соответствуют той же позиции, что и трехмерные координаты. В данном случае, так как последний компонент вычисляется как сумма $w1+w2$, чтобы гарантировать равенство результата $w1+w2$ значению 1.0 необходимо в компоненте w передать значение 0.0 (четвертый параметр метода `gl.uniform4f()`).

Наконец, вызов `gl.drawArrays(gl.TRIANGLES, 0, n)` в строке 58 выполняет вершинный шейдер. На каждом прогоне шейдер выполняет следующие три операции:

1. Координаты текущей вершины переписываются в переменную `a_Position`.
2. Значение `u_Translation` складывается со значением переменной `a_Position`.
3. Результат присваивается переменной `gl_Position`.

После выполнения вершинного шейдера поставленную цель можно считать достигнутой, потому что координаты всех вершин будут изменены и перемещенная фигура (в данном случае треугольник) будет отображена на экране. Если теперь загрузить файл `TranslatedTriangle.html` в браузер, вы увидите перемещенный треугольник.

После знакомства с операцией перемещения перейдем к операции вращения. В своей основе, реализация вращения мало чем отличается от реализации перемещения – она точно так же требует выполнения операций с координатами в вершинном шейдере.

Вращение

Вращение выполняется немного сложнее, чем перемещение, потому что требует указать больше элементов информации. Чтобы выполнить вращение необходимо определить:

- оси вращения (оси, относительно которых будет вращаться фигура);
- направление вращения (по часовой стрелке или против часовой стрелки);
- угол поворота (число градусов, на которое требуется повернуть фигуру).

Чтобы упростить описание, в этом разделе будет предполагаться, что фигуру требуется повернуть относительно оси Z, против часовой стрелки на угол β градусов. Приемы, что будут описаны здесь, с тем же успехом могут применяться для вращения относительно осей X и Y.

Если величина угла поворота β выражается положительным значением, считается, что вращение выполняется против часовой стрелки, если смотреть вдоль оси вращения в направлении отрицательных значений (см. рис. 3.21); такое вращение называют **положительным вращением**. По аналогии с системой координат, ваши руки могут помочь вам определить направление вращения. Если правой рукой изобразить систему координат так, чтобы большой палец, изображающий ось вращения, смотрел вам в лицо, согнутые пальцы будут показывать направление положительного вращения. Этот прием называют **правилom вращения правой руки**. Как отмечалось в главе 2, такая система координат используется в WebGL по умолчанию.

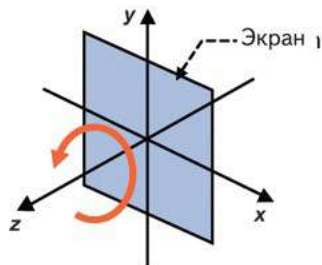


Рис. 3.21. Положительное вращение относительно оси Z

Теперь найдем выражение для вычисления результата вращения, по аналогии с примером реализации перемещения. Как показано на рис. 3.22, предполагается, что точка $p'(x', y', z')$ получается в результате поворота точки $p(x, y, z)$ на угол β относительно оси Z. Так как вращение выполняется относительно оси Z, координата Z не изменяется и ее можно пока игнорировать. Описание ниже содержит множество математических определений, поэтому будем двигаться вперед не спеша.

На рис. 3.22, r – это расстояние от начала координат до точки p , а α – угол между вектором Op и осью X. Эту информацию можно использовать для представления координат p , как показано в формуле 3.2.

Формула 3.2.

$$x = r \cos \alpha$$

$$y = r \sin \alpha$$

Аналогично можно выразить координаты точки p' через r , α и β , как показано ниже:

$$x' = r \cos (\alpha + \beta)$$

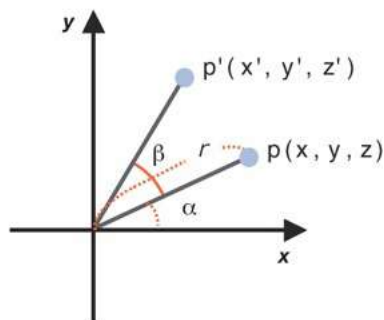


Рис. 3.22. Расчет новых координат точки, получающихся в результате вращения относительно оси Z

$$y' = r \sin(\alpha + \beta)$$

Применив теорему сложения тригонометрических функций², получаем следующее:

$$x' = r (\cos \alpha \cos \beta - \sin \alpha \sin \beta)$$

$$y' = r (\sin \alpha \cos \beta + \cos \alpha \sin \beta)$$

Наконец, подставив формулу 3.2 в эти выражения и сократив r и α , получаем формулу 3.3.

Формула 3.3.

$$x' = x \cos \beta - y \sin \beta$$

$$y' = x \sin \beta + y \cos \beta$$

$$z' = z$$

То есть, передав значения $\sin \beta$ и $\cos \beta$ в вершинный шейдер и выполнив в нем вычисления по формуле 3.3, мы получим координаты точки после поворота. Чтобы найти $\sin \beta$ и $\cos \beta$, можно воспользоваться методами JavaScript-объекта `Math`.

Давайте рассмотрим пример программы `RotatedTriangle`, осуществляющей поворот треугольника относительно оси Z, против часовой стрелки на угол 90 градусов. Результат работы программы показан на рис. 3.23.

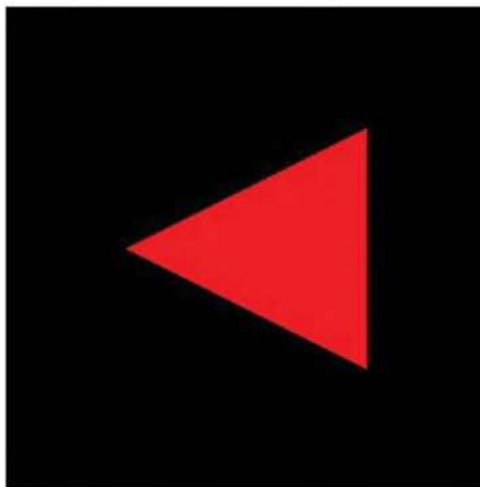


Рис. 3.23. `RotatedTriangle`

Пример программы (`RotatedTriangle.js`)

В листинге 3.5 представлено содержимое файла `RotatedTriangle.js` с исходным кодом программы, включающим измененный вершинный шейдер, осуществляющим операцию вращения. Фрагментный шейдер остался таким же, как в `TranslatedTriangle.js` и, как обычно, здесь не показан. И снова, для поддержки

² $\sin(a \pm b) = \sin a \cos b \pm \cos a \sin b$; $\cos(a \pm b) = \cos a \cos b \pm \sin a \sin b$.

изменений в вершинном шейдере, в функцию `main()` добавлено выполнение нескольких дополнительных шагов. Кроме того, в строках с 4 по 6 приводится комментарий с формулой 3.3 для напоминания.

Листинг 3.5. RotatedTriangle.js

```

1 // RotatedTriangle.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4     //  $x' = x \cos b - y \sin b$ 
5     //  $y' = x \sin b + y \cos b$ 
6     //  $z' = z$ 
7     'attribute vec4 a_Position;\n' +
8     'uniform float u_CosB, u_SinB;\n' +
9     'void main() {\n' +
10     '    gl_Position.x = a_Position.x * u_CosB - a_Position.y * u_SinB;\n' +
11     '    gl_Position.y = a_Position.x * u_SinB + a_Position.y * u_CosB;\n' +
12     '    gl_Position.z = a_Position.z;\n' +
13     '    gl_Position.w = 1.0;\n' +
14     '}\n';
15
16 // Фрагментный шейдер
17 ...
18
19 // Угол поворота
20 var ANGLE = 90.0;
21
22 function main() {
23     ...
24     // Определить координаты вершин
25     var n = initVertexBuffers(gl);
26     ...
27     // Передать в вершинный шейдер данные, необходимые для поворота фигуры
28     var radian = Math.PI * ANGLE / 180.0; // Преобразовать в радианы
29     var cosB = Math.cos(radian);
30     var sinB = Math.sin(radian);
31
32     var u_CosB = gl.getUniformLocation(gl.program, 'u_CosB');
33     var u_SinB = gl.getUniformLocation(gl.program, 'u_SinB');
34
35     gl.uniform1f(u_CosB, cosB);
36     gl.uniform1f(u_SinB, sinB);
37
38     // Указать цвет для очистки области рисования <canvas>
39     ...
40     // Нарисовать треугольник
41     gl.drawArrays(gl.TRIANGLES, 0, n);
42 }
43
44 function initVertexBuffers(gl) {
45     var vertices = new Float32Array([
46         0.0, 0.5, -0.5, -0.5, 0.5, -0.5
47     ]);
48     var n = 3; // Число вершин

```



```
...
105 return n;
106 }
```

Рассмотрим вершинный шейдер поближе. В нем нет ничего сложного:

```
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   //  $x' = x \cos b - y \sin b$ 
5   //  $y' = x \sin b + y \cos b$ 
6   //  $z' = z$ 
7   'attribute vec4 a_Position;\n' +
8   'uniform float u_CosB, u_SinB;\n' +
9   'void main() {\n' +
10  '   gl_Position.x = a_Position.x * u_CosB - a_Position.y * u_SinB;\n' +
11  '   gl_Position.y = a_Position.x * u_SinB + a_Position.y * u_CosB;\n' +
12  '   gl_Position.z = a_Position.z;\n' +
13  '   gl_Position.w = 1.0;\n' +
14  '}\n';
```

Поскольку целью является поворот треугольника на 90 градусов, необходимо вычислить синус и косинус этого угла. В строке 8 определяются две *uniform*-переменные, через которые вершинный шейдер будет принимать эти значения, вычисленные программой на JavaScript.

Можно было бы передать вершинному шейдеру величину угла в градусах и вычислять синус и косинус в самом шейдере. Однако, поскольку они остаются одинаковыми для всех вершин, эффективнее будет вычислить их один раз в программном коде на JavaScript.

Имена *uniform*-переменных, *u_CosB* и *u_SinB*, выбраны в соответствии с соглашениями, принятыми в данной книге. Как вы наверняка помните, *uniform*-переменные используются по той простой причине, что их значения остаются общими для всех вершин (то есть, не изменяются).

Как и в предыдущих примерах программ, компоненты вершин — *x*, *y*, *z* и *w* — передаются в вершинный шейдер в виде группы, посредством переменной-атрибута *a_Position*. Чтобы применить формулу 3.3 к координатам *X*, *Y* и *Z*, нужно обратиться к каждому компоненту переменной *a_Position* отдельно. Сделать это можно с помощью оператора «точка» (*.*), например: *a_Position.x*, *a_Position.y* и *a_Position.z* (см. рис. 3.24 и главу 6).

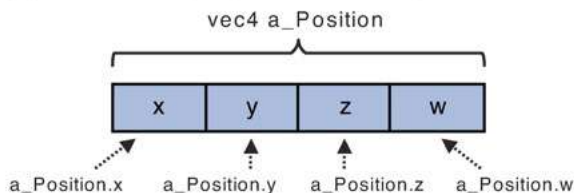


Рис. 3.24. Методы доступа к компонентам переменной типа *vec4*

Тот же самый оператор можно использовать для доступа к компонентам в переменной *gl_Position*, куда записываются координаты вершин, благодаря чему

вычисление выражения $x' = x \cos \beta - y \sin \beta$ из формулы 3.3 можно реализовать, как показано в строке 10:

```
10 ' gl_Position.x = a_Position.x * u_CosB - a_Position.y * u_SinB;\n' +
```

Аналогично вычисляется значение y' :

```
11 ' gl_Position.y = a_Position.x * u_SinB + a_Position.y * u_CosB;\n' +
```

Согласно формуле 3.3, координата Z просто переписывается в исходном виде (строка 12). Наконец, последнему компоненту w нужно присвоить значение 1.0:³

```
12 ' gl_Position.z = a_Position.z;\n' +
```

```
13 ' gl_Position.w = 1.0;\n' +
```

Теперь перейдем к функции `main()`, определение которой начинается в строке 25. Большая часть функции осталась прежней, как в программе `TranslatedTriangle.js`. Единственное отличие – передача $\cos \beta$ и $\sin \beta$ в вершинный шейдер. Для вычисления синуса и косинуса угла β можно воспользоваться методами `Math.sin()` и `Math.cos()`. Однако эти методы принимают величину угла в радианах, а не в градусах, поэтому нам нужно преобразовать градусы в радианы, умножив число градусов на число π и разделив произведение на 180. Число π доступно в виде свойства `Math.PI`, как показано в строке 50, где переменная `ANGLE` имеет значение 90 (градусов), как определено в строке 23:

```
50 var radian = Math.PI * ANGLE / 180.0; // Преобразовать в радианы
```

После преобразования в строках 51 и 52 вычисляются $\cos \beta$ и $\sin \beta$, и затем, в строках 60 и 61, полученные значения записываются в `uniform`-переменные:

```
51 var cosB = Math.cos(radian);
52 var sinB = Math.sin(radian);
53
54 var u_CosB = gl.getUniformLocation(gl.program, 'u_CosB');
55 var u_SinB = gl.getUniformLocation(gl.program, 'u_SinB');
...
60 gl.uniform1f(u_CosB, cosB);
61 gl.uniform1f(u_SinB, sinB);
```

После загрузки этой программы в браузер вы увидите на экране треугольник, повернутый на 90 градусов. Если переменной `ANGLE` присвоить отрицательное значение, треугольник будет повернут в противоположную сторону (по часовой стрелке). Например, чтобы повернуть треугольник по часовой стрелке, вы можете в строке 23 изменить число 90 на -90 , а все остальное за вас сделают `Math.cos()` и `Math.sin()`.

Для тех, кого волнуют проблемы эффективности и быстродействия, отметим, что выбранный подход (основанный на использовании двух `uniform`-переменных

³ В этой программе можно также выполнить присваивание `gl_Position.w = a_Position.w;`, потому что `a_Position.w` как раз имеет значение 1.0.

для передачи значений $\cos \beta$ и $\sin \beta$) не является оптимальным. Для передачи нескольких значений одновременно, можно было бы объявить одну `uniform`-переменную, как показано ниже:

```
uniform vec2 u_CosBSinB ;,
```

и передавать значения так:

```
gl.uniform2f( u_CosBSinB, cosB, sinB );
```

В этом случае получить переданные значения вершинном шейдере можно было бы с помощью выражений `u_CosBSinB.x` и `u_CosBSinB.y`.

Матрица преобразования: вращение

Формулы, представленные выше, с успехом можно использовать в простых случаях. Однако, по мере усложнения программы вы быстро обнаружите, что применение последовательностей формул требует слишком много усилий. Например, чтобы выполнить «перемещение после вращения», как показано на рис. 3.25, потребуется на основе формул 3.1 и 3.3 вывести еще одну формулу, описывающую нужную трансформацию, и затем реализовать ее в вершинном шейдере.

Однако выведение новых формул с последующей их реализацией в вершинных шейдерах требует немалых усилий. К счастью существует другой путь – применение **матриц преобразований** – который отлично подходит для управления компьютерной графикой.

Как показано на рис. 3.26, матрица – это прямоугольный массив чисел, расположенных строками (по горизонтали) и столбцами (по вертикали). Такая нотация упрощает запись вычислений, описанных в предыдущих разделах. Квадратные скобки указывают, что данные числа образуют единую группу.

Прежде чем переходить к обсуждению тонкостей использования матриц преобразований взамен формул, необходимо разобраться с операцией умножения матрицы на вектор. Вектор – это объект, представляющий кортеж из n чисел, таких как координаты вершины (0.0, 0.5, 1.0).

Умножение матрицы на вектор можно записать, как показано в формуле 3.4. (Оператор $\times$ умножения часто опускают, но мы решили оставить его для ясности.) Здесь новый вектор (слева) является результатом умножения матрицы (в центре) на исходный вектор (справа). Обратите внимание, что операция умножения матрицы на вектор не является коммутативной. Иными словами, результат выражения $A \times B$ будет отличаться от результата выражения $B \times A$. Подробнее на этой особенности мы остановимся в главе 6.

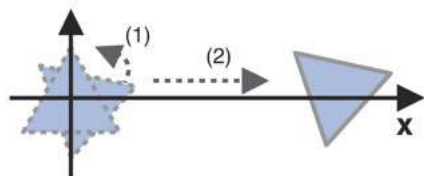


Рис. 3.25. Вращение треугольника с последующим перемещением

$$\begin{bmatrix} 8 & 3 & 0 \\ 4 & 3 & 6 \\ 3 & 2 & 6 \end{bmatrix}$$

Рис. 3.26. Пример матрицы

Формула 3.4.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Эта матрица имеет три строки и три столбца и называется матрицей 3×3 . Справа в выражении находится вектор, состоящий из трех элементов: x , y и z . (В случае умножения матрицы на вектор, вектор записывается вертикально, но он сохраняет тот же смысл, что и при записи по горизонтали.) Данный вектор, состоящий из трех компонентов, называется трехмерным вектором. И снова, квадратные скобки с обеих сторон от массива чисел (вектора) служат лишь для обозначения, что эти числа составляют единую группу.

Здесь значения x' , y' и z' определяются на основе значений элементов матрицы и вектора, как показано в формуле 3.5. Обратите внимание, что произведение матрицы на вектор может быть определено, только если число столбцов в матрице совпадает с числом строк в векторе.

Формула 3.5.

$$x' = ax + by + cz$$

$$y' = dx + ey + fz$$

$$z' = gx + hy + iz$$

Теперь, чтобы понять, как произведение матрицы на вектор может заменить формулы, рассматривавшиеся нами первоначально, сравним формулу 3.5 с формулой 3.3 (ниже она приводится еще раз, как формула 3.6).

Формула 3.6.

$$x' = x \cos \beta - y \sin \beta$$

$$y' = x \sin \beta + y \cos \beta$$

$$z' = z$$

Например, сравним вычисление x' :

$$x' = ax + by + cz$$

$$x' = x \cos \beta - y \sin \beta$$

В данном случае, если принять, что: $a = \cos \beta$, $b = -\sin \beta$ и $c = 0$, выражения станут идентичными. Аналогично сравним вычисление y' :

$$y' = dx + ey + fz$$

$$y' = x \sin \beta + y \cos \beta$$

В данном случае, если определить тождества: $d = \sin \beta$, $e = \cos \beta$ и $f = 0$, мы снова получим идентичные выражения. Последнее сравнение вычисления z' выглядит еще проще. Если определить тождества: $g = 0$, $h = 0$ и $i = 1$, мы вновь получим идентичные выражения.

Теперь, подставив тождества в формулу 3.4, получим формулу 3.7.

Формула 3.7.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos \beta & -\sin \beta & 0 \\ \sin \beta & \cos \beta & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Эта матрица называется **матрицей преобразований**, потому что она «преобразует» вектор-множитель (x, y, z) в вектор в левой части формулы (x', y', z') . Матрица преобразования, представляющая вращение, называется также **матрицей вращения**.

Как видите, элементами матрицы в формуле 3.7 является массив коэффициентов из формулы 3.6. Как только вы попривыкнете к матричной форме записи, вы сами убедитесь, что пользоваться матрицами проще, чем выражениями преобразований.

Как вы наверняка догадались, поскольку матрицы широко используются в трехмерной компьютерной графике, умножение матрицы и вектора легко реализуется в шейдерах. Однако, прежде чем увидеть, как это делается, давайте рассмотрим другие типы матриц преобразований, а затем уже начнем использовать их в шейдерах.

Матрица преобразования: перемещение

Очевидно, что если есть возможность использовать матрицу преобразования для представления вращения, должна быть возможность использовать матрицы и для других видов преобразований. Например, давайте сравним вычисление x' из формулы 3.1 и из формулы Equation 3.5:

$$x' = ax + by + cz \quad \text{--- Формула 3.5}$$

$$x' = x + T_x \quad \text{--- Формула 3.1}$$

Здесь во втором выражении имеется постоянный член T_x , а в первом его нет. Это означает, что нельзя получить аналог второго выражения с применением матрицы 3×3 . Зато такая возможность имеется с использованием матрицы 4×4 и четвертых компонентов в векторах координат (которые мы устанавливаем в 1) для ввода постоянных членов в уравнение. То есть, если предположить, что исходная точка p имеет координаты $(x, y, z, 1)$ и смещенная точка p' имеет координаты $(x', y', z', 1)$, мы получим следующую формулу 3.8.

Формула 3.8.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Это произведение вычисляется следующим образом:

Формула 3.9.

$$x' = ax + by + cz + d$$

$$y' = ex + fy + gz + h$$

$$z' = ix + jy + kz + l$$

$$1 = mx + ny + oz + p$$

Из тождества $1 = mx + ny + oz + p$ легко определить значения коэффициентов: $m = 0, n = 0, o = 0$ и $p = 1$. Кроме того, в этих выражениях присутствуют постоянные члены: d, h и l . Которые, похоже, могут пригодиться для приведения формулы 3.1 к матричному виду, потому что в ней тоже имеются постоянные члены. Давайте сравним формулу 3.9 с формулой 3.1, которую приведем еще раз:

$$x' = x + Tx$$

$$y' = y + Ty$$

$$z' = z + Tz$$

При сравнении выражений вычисления x' , можно заметить, что $a = 1, b = 0, c = 0$ и $d = T_x$. Аналогично, при сравнении выражений вычисления y' , можно заметить, что $e = 0, f = 1, g = 0$ и $h = T_y$; и, при сравнении выражений вычисления z' , можно заметить, что $i = 0, j = 0, k = 1$ и $l = T_z$. На основе этих результатов можно определить матрицу, описывающую перемещение, которая так и называется: **матрица перемещения** и показана в формуле 3.10.

Формула 3.10.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

И снова матрица вращения

К настоящему моменту мы сумели определить матрицы вращения и перемещения, эквивалентные формулам, использовавшимся в примерах программ выше. Теперь попробуем сделать последний шаг и объединить эти две матрицы; однако, матрица вращения (матрица 3×3) и матрица перемещения (матрица 4×4) имеют разное число элементов. К сожалению, матрицы разных размеров нельзя объединить, поэтому нам нужен некоторый способ привести их к одному размеру.

Для этого следует превратить матрицу вращения (матрицу 3×3) в матрицу 4×4 . Сделать это несложно, нужно лишь найти коэффициент для каждого выражения в формуле 3.9, сопоставив их с выражениями в формуле 3.3. Ниже приводятся обе формулы:

$$x' = x \cos \beta - y \sin \beta$$

$$y' = x \sin \beta + y \cos \beta$$

$$z' = z$$

$$x' = ax + by + cz + d$$

$$y' = ex + fy + gz + h$$

$$z' = ix + iy + kz + l$$

$$1 = mx + ny + oz + p$$

Например, сравнив $x' = x \cos \beta - y \sin \beta$ и $x' = ax + by + cz + d$, можно заметить, что $a = \cos \beta$, $b = -\sin \beta$, $c = 0$ и $d = 0$. Аналогично, после сравнения выражений вычисления y' и z' , мы получим матрицу вращения, представленную в формуле 3.11:

Формула 3.11.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \beta & -\sin \beta & 0 & 0 \\ \sin \beta & \cos \beta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Таким способом можно представить обе матрицы, вращения и перемещения, в виде единой матрицы 4×4 !

Пример программы (RotatedTriangle_Matrix.js)

Определив матрицу 4×4 , сделаем еще шаг и задействуем эту матрицу в программе WebGL, переписав программу RotatedTriangle, которая поворачивает треугольник на угол 90 градусов против часовой стрелки относительно оси Z с использованием матрицы вращения. В листинге 3.6 приводится содержимое файла RotatedTriangle_Matrix.js, результат, изображенный на рис. 3.23.

Листинг 3.6. RotatedTriangle_Matrix.js

```
1 // RotatedTriangle_Matrix.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'uniform mat4 u_xformMatrix;\n' +
6   'void main() {\n' +
7   '  gl_Position = u_xformMatrix * a_Position;\n' +
8   '}\n';
9
10 // Фрагментный шейдер
11 ...
12
13 // Угол поворота
14 var ANGLE = 90.0;
15
16 function main() {
17   ...
```

```

36 // Определить координаты вершин
37 var n = initVertexBuffers(gl);
38 ...
43 // Создать матрицу вращения
44 var radian = Math.PI * ANGLE / 180.0; // Преобразовать в радианы
45 var cosB = Math.cos(radian), sinB = Math.sin(radian);
46
47 // Примечание: в WebGL элементы следуют в порядке расположения по столбцам
48 var xformMatrix = new Float32Array([
49     cosB, sinB, 0.0, 0.0,
50     -sinB, cosB, 0.0, 0.0,
51     0.0, 0.0, 1.0, 0.0,
52     0.0, 0.0, 0.0, 1.0
53 ]);
54
55 // Передать матрицу вращения в вершинный шейдер
56 var u_xformMatrix = gl.getUniformLocation(gl.program, 'u_xformMatrix');
57 ...
61 gl.uniformMatrix4fv(u_xformMatrix, false, xformMatrix);
62
63 // Указать цвет для очистки области рисования <canvas>
64 ...
69 // Нарисовать треугольник
70 gl.drawArrays(gl.TRIANGLES, 0, n);
71 }
72
73 function initVertexBuffers(gl) {
74     var vertices = new Float32Array([
75         0.0, 0.5, -0.5, -0.5, 0.5, -0.5
76     ]);
77     var n = 3; // Число вершин
78     ...
105     return n;
106 }

```

Сначала рассмотрим вершинный шейдер:

```

2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4     'attribute vec4 a_Position;\n' +
5     'uniform mat4 u_xformMatrix;\n' +
6     'void main() {\n' +
7     '    gl_Position = u_xformMatrix * a_Position;\n' +
8     '}\n';

```

В строке 7 выполняется умножение матрицы вращения `u_xformMatrix`, описываемой формулой 3.11, на вектор `a_Position` с координатами вершины (вектор справа в формуле 3.11).

В примере программы `TranslatedTriangle` мы смогли выполнить сложение двух векторов в одной строке кода (`gl_Position = a_Position + u_Translation`). Аналогично, на языке GLSL ES можно в одной строке выполнить умножение матрицы на вектор. Это очень удобно, так как позволяет осуществить вычисление четырех выражений (формула 3.9) в одной строке. Данный пример еще раз

наглядно показывает, насколько язык GLSL ES оптимизирован под поддержку трехмерной компьютерной графики.

Так как матрица преобразования имеет размер 4×4 и язык GLSL ES требует определять типы для всех переменных, переменная `u_xformMatrix` в строке 5 определена с типом `mat4`. Как вы наверняка догадались, тип `mat4` представляет матрицу 4×4 .

Все остальные изменения в программе на JavaScript связаны с вычислениями матрицы вращения, в соответствии с формулой 3.11, и передачей ее в `u_xformMatrix`. Эта часть программы начинается со строки 44:

```

43 // Создать матрицу вращения
44 var radian = Math.PI * ANGLE / 180.0; // Преобразовать в радианы
45 var cosB = Math.cos(radian), sinB = Math.sin(radian);
46
47 // Примечание: в WebGL элементы следуют в порядке расположения по столбцам
48 var xformMatrix = new Float32Array([
49     cosB, sinB, 0.0, 0.0,
50     -sinB, cosB, 0.0, 0.0,
51     0.0, 0.0, 1.0, 0.0,
52     0.0, 0.0, 0.0, 1.0
53 ]);
54
55 // Передать матрицу вращения в вершинный шейдер
56 ...
61 gl.uniformMatrix4fv(u_xformMatrix, false, xformMatrix);

```

В строках 44 и 45 вычисляются значения косинуса и синуса, необходимые для матрицы вращения. Затем в строке 48 создается матрица `xformMatrix` типа `Float32Array`. В отличие от GLSL ES, в языке JavaScript отсутствует специальный тип для представления матриц, поэтому мы вынуждены использовать массив типа `Float32Array`. В связи с этим возникает один важный вопрос: в каком порядке должны следовать элементы в определении массива (в каком порядке должны следовать строки и столбцы в плоском, одномерном массиве). Существует два возможных порядка расположения: по строкам и по столбцам (см. рис. 3.27).

WebGL, как и OpenGL, требует, чтобы элементы матриц хранились в порядке расположения по столбцам. Так, например, матрица на рис. 3.27 должна храниться в массиве следующим образом: $[a, e, i, m, b, f, j, n, c, g, k, o, d, h, l, p]$. В примере программы (строки с 49 по 52) матрица вращения хранится в массиве типа `Float32Array` именно в таком порядке.

Созданный массив передается в `uniform`-переменную `u_xformMatrix` вызовом `gl.uniformMatrix4fv()` (строка 61). Обратите внимание на последнюю букву «v» в имени метода, которая указывает, что метод способен записать в переменную множество значений.

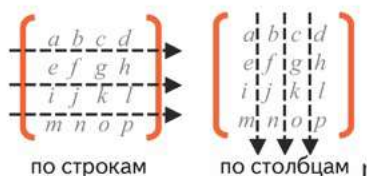


Рис. 3.27. Порядок расположения по строкам и по столбцам

gl.uniformMatrix4fv (location, transpose, array)

Записывает матрицу 4×4, находящуюся в массиве array, в uniform-переменную location.

Параметры	location	Определяет uniform-переменную, куда будет записана матрица.
	transpose	В WebGL должен иметь значение false.*
	array	Определяет массив с матрицей 4×4 с порядком расположения элементов по столбцам.
Возвращаемое значение	нет	
Ошибки	INVALID_OPERATION	Текущий объект программы не был скомпонован.
	INVALID_VALUE	Параметр transpose не равен false или длина массива меньше 16.

* Этот параметр определяет необходимость транспонирования матрицы. Операция транспонирования заключается во взаимной перестановке ее строк и столбцов (см. главу 7). Эта операция не поддерживается реализацией данного метода в WebGL, поэтому в данном параметре всегда следует передавать false.

После загрузки этой программы в браузер вы увидите на экране треугольник, повернутый на 90 градусов. Поздравляем! Теперь вы знаете, как с помощью матрицы преобразований повернуть треугольник.

Применение того же подхода для перемещения

Теперь, после знакомства с формулами 3.10 и 3.11, совершенно понятно, что в виде матрицы 4×4 можно представить и операцию вращения, и операцию перемещения. В обеих операциях используются матрицы в форме $\langle \text{новые координаты} \rangle = \langle \text{матрица преобразований} \rangle * \langle \text{исходные координаты} \rangle$. Это выражение реализовано в вершинном шейдере в строке 7:

```
7   ' gl_Position = u_xformMatrix * a_Position;\n' +
```

Из этого следует, что если матрицу вращения в массиве xformMatrix заменить матрицей перемещения, вершинный шейдер выполнит перемещение треугольника, в точности как это было сделано с помощью выражений (рис. 3.18).

Для этого нужно изменить строку 17 в файле RotatedTriangle_Matrix.js, подставив в нее расстояния перемещения из предыдущего примера:

```
17   varTx = 0.5, Ty = 0.5, Tz = 0.0;
```

Кроме того, необходимо переписать код, создающий матрицу, не забыв при этом, что матрица должна сохраняться в массиве в порядке расположения элементов по столбцам. Давайте сохраним имя переменной-массива прежним, xformMatrix, потому что это позволит нам использовать уже имеющийся код. Наконец, в этом примере не используется переменная ANGLE, поэтому строки с 43 по 45 следует закомментировать:

```

43 // Создать матрицу вращения
44 // var radian = Math.PI * ANGLE / 180.0; // Преобразовать в радианы
45 // var cosB = Math.cos(radian), sinB = Math.sin(radian);
46
47 // Примечание: в WebGL элементы следуют в порядке расположения по столбцам
48 var xformMatrix = new Float32Array([
49     1.0, 0.0, 0.0, 0.0,
50     0.0, 1.0, 0.0, 0.0,
51     0.0, 0.0, 1.0, 0.0,
52     Tx, Ty, Tz, 1.0
53 ]);

```

После внесения изменений запустите программу и вы увидите ту же картину, что показана на рис. 3.18. С помощью матрицы можно применять самые разные преобразования, используя тот же самый вершинный шейдер. Это объясняет, почему матрицы преобразований являются таким удобным и мощным инструментом создания трехмерной графики, и именно поэтому мы так подробно обсуждали данную тему в этой главе.

Матрица преобразования: масштабирование

В заключение давайте определим матрицу преобразования для масштабирования, взяв за основу ту же самую исходную точку p и ее положение p' после масштабирования.

Если допустить, что коэффициенты масштабирования по осям X , Y и Z имеют значения S_x , S_y и S_z , соответственно, мы приходим к следующим тождествам:

$$x' = S_x \times x$$

$$y' = S_y \times y$$

$$z' = S_z \times z$$

Сравнив их с формулой 3.9, мы получаем следующую матрицу преобразования:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Если записать эту матрицу в переменную `xformMatrix`, как это было сделано с матрицей перемещения в предыдущем примере, мы сможем выполнить масштабирование треугольника, используя все тот же вершинный шейдер из `RotatedTriangle_Matrix.js`. Например, следующая программа выполняет масштабирование треугольника по вертикали с коэффициентом 1.5, как показано на рис. 3.29:

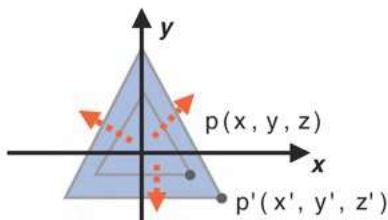


Рис. 3.28. Масштабирование


```
17 varSx = 1.0, Sy = 1.5, Sz = 1.0;
...
47 // Примечание: в WebGL элементы следуют в порядке расположения по столбцам
48 var xformMatrix = new Float32Array([
49     Sx, 0.0, 0.0, 0.0,
50     0.0, Sy, 0.0, 0.0,
51     0.0, 0.0, Sz, 0.0,
52     0.0, 0.0, 0.0, 1.0
53 ]
```

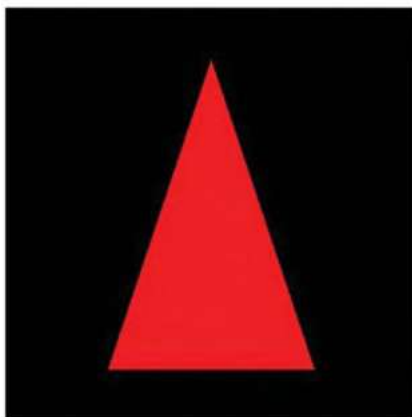


Рис. 3.29. Результат масштабирования треугольника по вертикали

Обратите внимание, что если коэффициент масштабирования S_x , S_y или S_z установить в значение 0.0, соответствующий размер после масштабирования получится равным 0.0. Если необходимо сохранить первоначальные размеры, коэффициенты масштабирования должны быть равны 1.

В заключение

В этой главе мы исследовали возможность передачи в вершинный шейдер множества единиц информации о вершинах, разные типы фигур, которые можно рисовать на основе этой информации, и приемы преобразований этих фигур. Здесь рассматривались разные фигуры, от точки до треугольника, но методы использования шейдеров остались теми же самыми, как и в предыдущей главе. Мы также познакомились с матрицами и узнали, как ими пользоваться для перемещения, вращения и масштабирования 2-мерных фигур. Несмотря на довольно высокую сложность рассматривавшихся тем, в настоящий момент вы должны неплохо понимать математический аппарат, лежащий в основе преобразований.

В следующей главе мы займемся исследованием более сложных преобразований, но при этом будем пользоваться вспомогательной библиотекой, чтобы скрыть за ее фасадом технические тонкости и все свое внимание сосредоточить на решении задач более высокого уровня.



ГЛАВА 4.

Дополнительные преобразования и простая анимация

В главе 3, «Рисование и преобразование треугольников», рассказывалось, как пользоваться буферными объектами для рисования более сложных фигур и как преобразовывать их с использованием простых выражений. Поскольку сложность такого подхода резко возрастает с увеличением числа преобразований, мы рассмотрели более простое решение, основанное на матрицах. В этой главе мы продолжим исследование преобразований и начнем с их объединения в анимационные эффекты. В частности, в этой главе мы:

- познакомимся с библиотекой, реализующей операции с матрицами преобразований и скрывающей за своим фасадом математические тонкости;
- посмотрим, как с ее помощью можно легко и быстро объединить несколько преобразований;
- исследуем приемы анимации и применение библиотеки для их реализации.

Все эти приемы образуют фундамент для конструирования еще более сложных WebGL-программ и будут использоваться в примерах программ в последующих главах.

Перемещение с последующим вращением

Как было показано в главе 3, такие преобразования, как перемещение, вращение и масштабирование, можно представить в виде матриц 4×4 , однако определение таких матриц вручную может сильно осложнить разработку WebGL-программ. Чтобы упростить эту задачу, многие разработчики пользуются вспомогательными библиотеками, автоматизирующими создание матриц и выполнение рутинных операций с ними. Для WebGL существует несколько подобных, общедоступных библиотек, но в этом разделе мы будем использовать библиотеку, специально созданную для данной книги, и посмотрим, как с ее помощью можно комбинировать преобразования для достижения таких результатов, как «перемещение треугольника с последующим его вращением».

Библиотека матричных преобразований: *cuon-matrix.js*

В спецификации OpenGL, базовой для спецификации WebGL, не требуется вручную определять каждый элемент матрицы преобразования перед ее использованием. Вместо этого OpenGL поддерживает множество вспомогательных функций, упрощающих конструирование матриц.

Например, функция `glTranslatef()`, принимающая расстояния перемещения по осям X, Y и Z в виде аргументов, создает и настраивает соответствующую матрицу преобразования, необходимую для перемещения фигуры (см. рис. 4.1).

`glTranslatef(5, 80, 30);` → $\begin{bmatrix} 1 & 0 & 0 & 5 \\ 0 & 1 & 0 & 80 \\ 0 & 0 & 1 & 30 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Рис. 4.1. Пример вызова функции `glTranslatef()` в OpenGL

К сожалению, спецификация WebGL не предусматривает реализацию этих функций, поэтому любому, кто пожелает использовать подобный подход, придется найти соответствующую библиотеку или реализовать требуемые функции самостоятельно. Поскольку эти функции дают массу удобств, мы создали для вас такую библиотеку на языке JavaScript (*cuon-matrix.js*), с помощью которой вы сможете создавать матрицы преобразований подобно тому, как это делается в OpenGL. Хотя эта библиотека и была написана специально для данной книги, она достаточно универсальна и вы бесплатно можете использовать ее в своих приложениях.

Как вы уже видели в главе 3, в программах на JavaScript тригонометрические функции доступны как методы объекта `Math`. Аналогично функции создания матриц преобразований доступны как методы объекта `Matrix4`, который определяется в файле *cuon-matrix.js*. `Matrix4` – это новый объект и, как следует из его имени, реализует операции с матрицами 4×4 . Эти матрицы представлены типизированными массивами `Float32Array`, с которыми мы уже встречались в главе 3.

Чтобы получить представление о возможностях библиотеки, давайте перепишем программу `RotatedTriangle_Matrix`, задействовав в ней объект `Matrix4` и его методы. Новая программа называется `RotatedTriangle_Matrix4`.

Так как объект `Matrix4` определен в библиотеке *cuon-matrix.js*, ее необходимо подключить в файле HTML перед использованием объекта. Для этого достаточно просто добавить тег `<script>`, как показано в листинге 4.1.

Листинг 4.1. `RotatedTriangle_Matrix4.html` (код загрузки библиотеки)

```
13 <script src="../../lib/webgl-debug.js"></script>
14 <script src="../../lib/cuon-utils.js"></script>
15 <script src="../../lib/cuon-matrix.js"></script>
16 <script src="RotatedTriangle_Matrix4.js"></script>
```

А теперь давайте посмотрим, как пользоваться библиотекой и сравним предыдущую программу `RotatedTriangle_Matrix.js` с программой `RotatedTriangle_Matrix4.js`, использующей объект `Matrix4`.

Пример программы (RotatedTriangle_Matrix4.js)

Эта программа почти полностью повторяет программу `RotatedTriangle_Matrix.js` из главы 3, за исключением реализации нескольких новых шагов, связанных с созданием матрицы преобразования и передачи ее в вершинный шейдер через переменную `u_xformMatrix`.

Ниже показано, как в `RotatedTriangle_Matrix.js` создается матрица преобразования:

```
1 // RotatedTriangle_Matrix.js
...
43 // Создать матрицу вращения
44 var radian = Math.PI * ANGLE / 180.0; // Преобразовать в радианы
45 var cosB = Math.cos(radian), sinB = Math.sin(radian);
46
47 // Примечание: в WebGL элементы следуют в порядке расположения по столбцам
48 var xformMatrix = new Float32Array([
49     cosB, sinB, 0.0, 0.0,
50     -sinB, cosB, 0.0, 0.0,
51     0.0, 0.0, 1.0, 0.0,
52     0.0, 0.0, 0.0, 1.0
53 ]);
...
61 gl.uniformMatrix4fv(u_xformMatrix, false, xformMatrix);
```

Нам нужно переписать эту часть программы, задействовав объект `Matrix4` и его метод `setRotate()` для создания матрицы вращения. Ниже приводится переделанный фрагмент из `RotatedTriangle_Matrix4.js`:

```
1 // RotatedTriangle_Matrix4.js
...
47 // Создать объект Matrix4
48 var xformMatrix = new Matrix4();
49 // Создать матрицу вращения в переменной xformMatrix
50 xformMatrix.setRotate(ANGLE, 0, 0, 1);
...
56 // Передать полученную матрицу вращения в вершинный шейдер
57 gl.uniformMatrix4fv(u_xformMatrix, false, xformMatrix.elements);
```

Как видите, в строках 48 и 50 создается требуемая матрица, которая затем, как в предыдущем примере, присваивается `uniform-переменной`. Объект `Matrix4` создается с помощью оператора `new`, так же как обычные JavaScript-объекты `Array` или `Date`. Сначала в строке 48 создается объект `Matrix4`, затем, в строке 50, вызовом его метода `setRotate()` заполняется матрица вращения, которая потом записывается в объект `xformMatrix`.

Метод `setRotate()` принимает четыре параметра: угол поворота (в градусах, а не в радианах) и оси, относительно которых выполняется поворот. Выбор осей вращения определяется параметрами `x`, `y` и `z`, а направление определяется линией, соединяющей точки с координатами `(0, 0, 0)` и `(x, y, z)`. Как вы наверняка помните из главы 3 (рис. 3.21), угол поворота положителен, если вращение выполняется

против часовой стрелки. В данном примере, так как вращение выполняется относительно оси Z, параметры, соответствующие осям, установлены как (0, 0, 1):

```
50 xformMatrix.setRotate(ANGLE, 0, 0, 1);
```

Аналогично, чтобы выполнить поворот относительно оси X, нужно передать параметры $x = 1$, $y = 0$ и $z = 0$, а чтобы выполнить поворот относительно оси Y – параметры $x = 0$, $y = 1$ и $z = 0$. После подготовки матрицы вращения в `xformMatrix`, ее остается только передать в вершинный шейдер, вызвав уже знакомый нам метод `gl.uniformMatrix4fv()`. Но, обратите внимание, что мы не можем передать этому методу объект `Matrix4` непосредственно, потому что он ожидает получить типизированный массив. Поэтому мы использовали свойство `elements` объекта `Matrix4`, возвращающее матрицу в виде такого типизированного массива:

```
57 gl.uniformMatrix4fv(u_xformMatrix, false, xformMatrix.elements);
```

В табл. 4.1 перечислены свойства и методы, поддерживаемые объектом `Matrix4`.

Таблица 4.1. Свойства и методы, поддерживаемые объектом `Matrix4`

Свойства и методы	Описание
<code>Matrix4.setIdentity()</code>	Инициализирует объект как единичную матрицу.*
<code>Matrix4.setTranslate(x, y, z)</code>	Заполняет объект <code>Matrix4</code> как матрицу перемещения, где параметр x определяет расстояние перемещения по оси X, параметр y – расстояние перемещения по оси Y, и параметр z – расстояние перемещения по оси Z.
<code>Matrix4.setRotate(angle, x, y, z)</code>	Заполняет объект <code>Matrix4</code> как матрицу вращения, представляющую поворот на угол <code>angle</code> (в градусах) относительно осей (x, y, z) . Координаты (x, y, z) не требуется нормализовать . (См. главу 8, «Освещение объектов».)
<code>Matrix4.setScale(x, y, z)</code>	Заполняет объект <code>Matrix4</code> как матрицу масштабирования с коэффициентами x , y и z .
<code>Matrix4.translate(x, y, z)</code>	Умножает матрицу, хранящуюся в объекте <code>Matrix4</code> , на матрицу перемещения, представляющую перемещение, где параметр x определяет расстояние перемещения по оси X, параметр y – расстояние перемещения по оси Y, и параметр z – расстояние перемещения по оси Z. Результат сохраняется в объекте <code>Matrix4</code> .
<code>Matrix4.rotate(angle, x, y, z)</code>	Умножает матрицу, хранящуюся в объекте <code>Matrix4</code> , на матрицу перемещения, представляющую поворот на угол <code>angle</code> (в градусах) относительно осей (x, y, z) . Результат сохраняется в объекте <code>Matrix4</code> . Координаты (x, y, z) не требуется нормализовать. (См. главу 8, «Освещение объектов».)

Свойства и методы	Описание
<code>Matrix4.scale(x, y, z)</code>	Умножает матрицу, хранящуюся в объекте <code>Matrix4</code> , на матрицу перемещения, представляющую масштабирования с коэффициентами <code>x</code> , <code>y</code> и <code>z</code> . Результат сохраняется в объекте <code>Matrix4</code> .
<code>Matrix4.set(m)</code>	Переписывает содержимое матрицы <code>m</code> в объект <code>Matrix4</code> . Матрица <code>m</code> сама должна быть объектом <code>Matrix4</code> .
<code>Matrix4.elements</code>	Типизированный массив (<code>Float32Array</code>) с элементами их матрицы, хранящейся в объекте <code>Matrix4</code> .

* Единичная матрица – это матрица, которая в операции умножения матриц действует подобно скалярному значению 1 в операции умножения чисел. Результатом умножения матрицы на единичную матрицу является исходная матрица. В единичной матрице элементы главной диагонали имеют значение 1.0, а остальные – значение 0.0.

Как показано в табл. 4.1, объект `Matrix4` поддерживает две группы методов: с именами, начинающимися с префикса `set`, и все остальные. Методы с именами, начинающимися с префикса `set`, создают матрицу преобразований на основе аргументов и сохраняют ее в объекте `Matrix4`. Остальные методы, напротив, умножают матрицу, хранящуюся в объекте `Matrix4`, на матрицу, которая должна быть создана на основе аргументов, и сохраняют результат обратно в объект `Matrix4`.

Методы, перечисленные в таблице, одновременно и мощные, и гибкие. Но, что самое важное, они существенно упрощают изменение матриц преобразований. Например, если вам потребуется вместо вращения выполнить перемещение треугольника, достаточно будет изменить лишь строку 50, как показано ниже:

```
50  xformMatrix.setTranslate( 0.5, 0.5, 0.0);
```

В данном примере переменная с именем `xformMatrix` представляет обобщенную матрицу преобразования. В своих приложениях вы можете использовать любые другие имена (например, `rotMatrix`, как в главе 3).

Объединение нескольких преобразований

Теперь, когда вы познакомились с объектом `Matrix4`, посмотрим, как можно использовать его для объединения двух преобразований: перемещения с последующим вращением. Именно это делает следующий пример программы `RotatedTranslatedTriangle`, результат работы которой показан на рис. 4.2. Обратите внимание, что в этой программе используется более маленький треугольник, чтобы проще было понять эффект преобразования.

Очевидно, что этот пример выполняет следующие два преобразования, изображенные на рис. 4.3:

1. Перемещает треугольник по оси X (1).
2. Выполняет поворот по оси Z (2).



Рис. 4.2. RotatedTranslatedTriangle

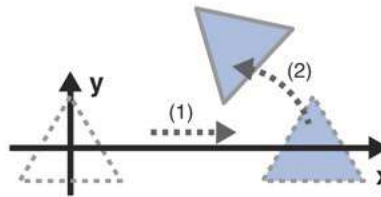


Рис. 4.3. Перемещение и поворот треугольника

На основе описания выше, формулу перемещения (1) треугольника можно записать так:

Формула 4.1.

$$\langle \text{координаты после перемещения} \rangle = \langle \text{матрица перемещения} \rangle \times \langle \text{исходные координаты} \rangle$$

Затем *<координаты после перемещения>* нужно повернуть (2):

Формула 4.2.

$$\langle \text{координаты после перемещения и вращения} \rangle = \langle \text{матрица вращения} \rangle \times \langle \text{координаты после перемещения} \rangle$$

Вычисления по этим формулам можно было бы выполнить по отдельности, однако их можно объединить, подставив формулу 4.1 в формулу 4.2.

Формула 4.3.

$$\langle \text{координаты после перемещения и вращения} \rangle = \langle \text{матрица вращения} \rangle \times (\langle \text{матрица перемещения} \rangle \times \langle \text{исходные координаты} \rangle)$$

где выражение

$$\langle \text{матрица вращения} \rangle \times (\langle \text{матрица перемещения} \rangle \times \langle \text{исходные координаты} \rangle)$$

эквивалентно выражению

$$(\langle \text{матрица вращения} \rangle \times \langle \text{матрица перемещения} \rangle) \times \langle \text{исходные координаты} \rangle$$

Этот заключительный шаг, вычисление выражения $\langle \text{матрица вращения} \rangle \times \langle \text{матрица перемещения} \rangle$, можно выполнить в программе на JavaScript и передать в вершинный шейдер готовый результат. Объединение нескольких преобразований, как в данном примере, называют **моделированием преобразований** (или **моделью преобразований**), а матрицу, представляющую такую модель преобразований, называют **матрицей модели**.

В качестве напоминания посмотрим еще раз, как выполняется умножение матриц, которые определены так:

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}, \quad B = \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix}$$

Произведение матриц A и B, имеющих размер 3×3, находится следующим образом:

Формула 4.4.

$$\begin{bmatrix} a_{00} \times b_{00} \times a_{01} \times b_{10} \times a_{02} \times b_{20} & a_{00} \times b_{01} \times a_{01} \times b_{11} \times a_{02} \times b_{21} & a_{00} \times b_{02} \times a_{01} \times b_{12} \times a_{02} \times b_{22} \\ a_{10} \times b_{00} \times a_{11} \times b_{10} \times a_{12} \times b_{20} & a_{10} \times b_{01} \times a_{11} \times b_{11} \times a_{12} \times b_{21} & a_{10} \times b_{02} \times a_{11} \times b_{12} \times a_{12} \times b_{22} \\ a_{20} \times b_{00} \times a_{21} \times b_{10} \times a_{22} \times b_{20} & a_{20} \times b_{01} \times a_{21} \times b_{11} \times a_{22} \times b_{21} & a_{20} \times b_{02} \times a_{21} \times b_{12} \times a_{22} \times b_{22} \end{bmatrix}$$

Мы использовали для примера матрицы 3×3, однако тот же самый подход можно распространить на более привычные матрицы 4×4. Тем не менее, обратите внимание, что порядок умножения матриц играет важную роль. Результат A * B не равен B * A.

Как вы уже наверняка догадались, библиотека `cuon-matrix.js` поддерживает метод, реализующий умножение объектов `Matrix4`. Давайте посмотрим, как пользоваться этим методом для объединения матриц перемещения и вращения в единую матрицу преобразования.

Пример программы (*RotatedTranslatedTriangle.js*)

В листинге 4.2 приводится содержимое файла `RotatedTranslatedTriangle.js`. Вершинный и фрагментный шейдеры в нем остались такими же, как в программе `RotatedTriangle_Matrix4.js`, рассматривавшейся в предыдущем разделе, за исключением имени `uniform`-переменной, которое изменилось с `u_xformMatrix` на `u_ModelMatrix`.

Листинг 4.2. `RotatedTranslatedTriangle.js`

```
1 // RotatedTranslatedTriangle.js
```

```

2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'uniform mat4 u_ModelMatrix;\n' +
6   'void main() {\n' +
7     '  gl_Position = u_ModelMatrix * a_Position;\n' +
8     '}\n';
9 // Фрагментный шейдер
...
16 function main() {
    ...
33   // Определить координаты вершин
34   var n = initVertexBuffers(gl);
    ...
40   // Создать объект Matrix4 для модели преобразования
41   var modelMatrix = new Matrix4();
42
43   // Создать матрицу модели
44   var ANGLE = 60.0; // Угол поворота
45   var Tx = 0.5;     // Расстояние перемещения
46   modelMatrix.setRotate(ANGLE, 0, 0, 1); // Определить матрицу вращения
47   modelMatrix.translate(Tx, 0, 0); // Умножить modelMatrix на
                                     ➡ матрицу перемещения
48
49   // Передать матрицу модели в вершинный шейдер
50   var u_ModelMatrix = gl.getUniformLocation(gl.program, ' u_ModelMatrix');
    ...
56   gl.uniformMatrix4fv(u_ModelMatrix, false, modelMatrix.elements);
    ...
63   // Нарисовать треугольник
64   gl.drawArrays(gl.TRIANGLES, 0, n);
65 }
66
67 function initVertexBuffers(gl) {
68   var vertices = new Float32Array([
69     0.0, 0.3, -0.3, -0.3, 0.3, -0.3
70   ]);
71   var n = 3; // Число вершин
    ...
99   return n;
100 }

```

Ключевыми в листинге 4.2 являются строки 46 и 47, где вычисляется произведение *<матрица вращения>* × *<матрица перемещения>*:

```

46 modelMatrix.setRotate(ANGLE, 0, 0, 1); // Определить матрицу вращения
47 modelMatrix.translate(Tx, 0, 0); // Умножить modelMatrix на
                                     ➡ матрицу перемещения

```

В строке 46 вызывается метод, начинающийся с префикса `set` (`setRotate()`), поэтому полученная в результате матрица вращения записывается в переменную `modelMatrix`. В следующей строке (47) вызывается метод, не начинающийся с префикса `set` (`translate()`), который, как описывалось выше, создает матрицу

перемещения на основе своих аргументов и умножает на нее матрицу, хранящуюся в объекте `modelMatrix`, а результат записывает обратно в `modelMatrix`. То есть, если `modelMatrix` уже содержит матрицу вращения, этот метод вычислит произведение $\langle \text{матрица вращения} \rangle \times \langle \text{матрица перемещения} \rangle$ и сохранит результат в `modelMatrix`.

Возможно вы обратили внимание, что порядок следования слов в заголовке «Перемещение с последующим вращением» противоположен порядку следования матриц в выражении $\langle \text{матрица вращения} \rangle \times \langle \text{матрица перемещения} \rangle$. Как показано в формуле 4.3, это обусловлено тем, что матрица преобразования умножается на вектор с исходными координатами треугольника.

Результат вычислений передается в вершинный шейдер через переменную `u_ModelMatrix`, в строке 56, а затем выполняется обычная операция рисования (строка 64). Если теперь загрузить эту программу в браузер, вы увидите красный треугольник, который сначала был перемещен вдоль оси X, а затем повернут относительно оси Z.

Эксперименты с примером программы

Давайте перепишем программу так, чтобы она сначала выполняла поворот, а потом перемещение. Для этого требуется всего лишь поменять местами операции вращения и перемещения. При этом, создание матрицы перемещения следует выполнить с помощью `set`-метода `setTranslate()`:

```
46 modelMatrix.setTranslate(Tx, 0, 0);  
47 modelMatrix.rotate(ANGLE, 0, 0, 1);
```

Результат работы этой программы показан на рис. 4.4.



Рис. 4.4. Поворот и перемещение треугольника

Как видите, изменив порядок операций вращения и перемещения, мы получили другой результат. Причина этого станет очевидна, если вы посмотрите на рис. 4.5.

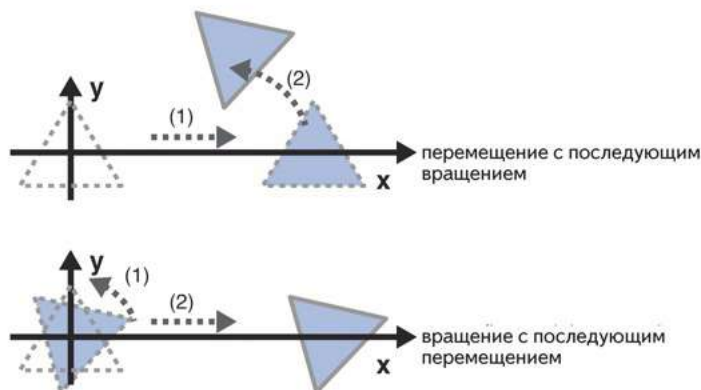


Рис. 4.5. Изменение порядка преобразований приводит к другому результату

На этом мы завершаем первое знакомство с методами, объявленными в библиотеке `cuon-matrix.js` и предназначенными для создания матриц преобразований. Мы будем пользоваться ими на протяжении всей оставшейся части книги, поэтому у вас еще не раз появится возможность исследовать их поведение.

Анимация

До сих пор в этой главе рассказывалось, как осуществлять преобразования фигур с помощью библиотеки матричных операций. Теперь вы достаточно хорошо знакомы с WebGL, чтобы сделать следующий шаг и применить имеющиеся знания для реализации анимационных эффектов.

Начнем с создания примера программы `RotatingTriangle`, которая непрерывно вращает треугольник с постоянной скоростью (45 градусов/сек). На рис. 4.6 показано несколько наложенных друг на друга изображений экрана, воспроизводимых программой `RotatingTriangle`, чтобы вы могли понять, как осуществляется вращение.

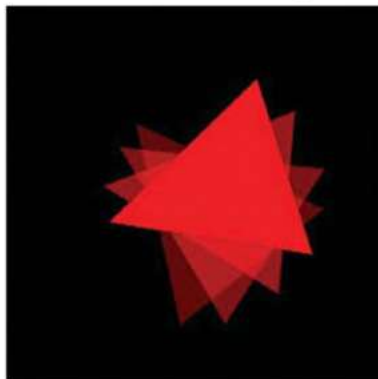


Рис. 4.6. Несколько перекрывающихся изображений экрана, воспроизводимых программой `RotatingTriangle`

Основы анимации

Чтобы воспроизвести эффект вращения треугольника, нужно просто перерисовать треугольник, немного поворачивая его каждый раз.

На рис. 4.7 показаны отдельные треугольники, которые рисуются в моменты времени t_0 , t_1 , t_2 , t_3 и t_4 . Каждый треугольник все еще остается всего лишь статическим изображением, но при этом все они повернуты на разные углы, по сравнению с исходным положением. Если вы увидите последовательную смену этих изображений, ваш мозг обнаружит изменения и сгладит границы между ними, породив ощущение поворачивания треугольника. Конечно, перед рисованием нового треугольника необходимо очистить область на экране. (Именно поэтому нужно вызывать `gl.clear()` перед рисованием чего бы то ни было.) Такой способ анимации можно применять не только к 2-мерным фигурам, но и к трехмерным объектам.

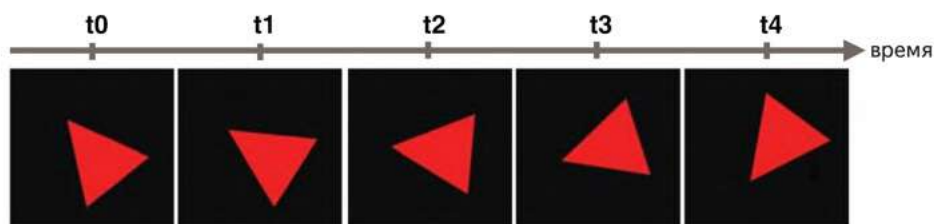


Рис. 4.7. Каждый раз рисуется треугольник, повернутый на больший угол

Для воспроизведения анимационного эффекта таким способом требуется два ключевых механизма:

Механизм 1: который позволил бы снова и снова вызывать функцию рисования треугольника в моменты времени t_0 , t_1 , t_2 , t_3 и так далее.

Механизм 2: осуществляющий очистку области рисования новым треугольником.

Если со вторым механизмом все просто – мы уже знаем, как это сделать, то с первым механизмом дело обстоит несколько сложнее, поэтому давайте шаг за шагом исследуем пример программы.

Пример программы (*RotatingTriangle.js*)

В листинге 4.3 приводится содержимое файла `RotatingTriangle.js`. Вершинный и фрагментный шейдеры в нем остались такими же, как в предыдущем примере. Однако здесь мы включили вершинный шейдер в листинг, чтобы показать выполнение умножения матрицы преобразования на координаты вершины.

Ниже перечислены три основных отличия от предыдущего примера:

- поскольку программа должна рисовать треугольник многократно, она была изменена так, что цвет для очистки области рисования теперь определяется один раз за все время выполнения программы (строка 44), а не перед опе-

рацией рисования; не забывайте, что цвет очистки запоминается системой WebGL, пока не будет задан другой цвет;

- фактический механизм повторного вызова [Механизм 1] функции рисования добавлен в строках с 59 по 64;
- второй механизм [Механизм 2], осуществляющий очистку и рисование, треугольника, определен как единая функция (`draw()`, в строке 102).

Эти различия выделены в листинге 4.3 (они отмечены метками (1), (2) и (3)). Давайте рассмотрим их подробнее.

Листинг 4.3. RotatingTriangle.js

```

1 // RotatingTriangle.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'uniform mat4 u_ModelMatrix;\n' +
6   'void main() {\n' +
7   '   gl_Position = u_ModelMatrix * a_Position;\n' +
8   '}\n';
9 // Фрагментный шейдер
10 ...
16 // Угол поворота (град/сек)
17 var ANGLE_STEP = 45.0;
18
19 function main() {
20   ...
21   // Определить координаты вершин
22   var n = initVertexBuffers(gl);
23   ...
24   // Указать цвет для очистки области рисования <canvas>      <- (1)
25   gl.clearColor(0.0, 0.0, 0.0, 1.0);
26
27   // Получить ссылку на переменную u_ModelMatrix
28   var u_ModelMatrix = gl.getUniformLocation(gl.program, ' u_ModelMatrix');
29   ...
30   // Текущий угол поворота треугольника
31   var currentAngle = 0.0;
32   // Объект Matrix4 для модели преобразования
33   var modelMatrix = new Matrix4();
34
35   // Начать рисование треугольника      <- (2)
36   var tick = function() {
37     currentAngle = animate(currentAngle); // Изменить угол поворота
38     draw(gl, n, currentAngle, modelMatrix, u_ModelMatrix);
39     requestAnimationFrame(tick); // Потребовать от браузера вызвать tick
40   };
41   tick();
42 }
43
44 function initVertexBuffers(gl) {
45   var vertices = new Float32Array ([
46     0.0, 0.5, -0.5, -0.5, 0.5, -0.5
47   ]

```



```

70  });
71  var n = 3; // Число вершин
    ...
96  return n;
97  }
98
99  function draw(gl,n, currentAngle, modelMatrix, u_ModelMatrix){      <--(3)
100  // Определить матрицу вращения
101  modelMatrix.setRotate(currentAngle, 0, 0, 1);
102
103  // Передать матрицу в вершинный шейдер
104  gl.uniformMatrix4fv( u_ModelMatrix, false, modelMatrix.elements);
105
106  // Очистить <canvas>
107  gl.clear(gl.COLOR_BUFFER_BIT);
108
109  // Нарисовать треугольник
110  gl.drawArrays(gl.TRIANGLES, 0, n);
111  }
112
113  // Момент времени, когда функция была вызвана в последний раз
114  var g_last = Date.now();
115  function animate(angle) {
116  // Вычислить прошедшее время
117  var now = Date.now();
118  var elapsed = now - g_last; // в миллисекундах
119  g_last = now;
120  // Изменить текущий угол поворота (в соответствии с прошедшим временем)
121  var newAngle = angle + (ANGLE_STEP * elapsed) / 1000.0;
122  return newAngle % 360;
123  }

```

В строке 7, в вершинном шейдере, выполняется умножение матрицы преобразования на координаты вершины, как это уже делалось в программе `RotatedTranslatedTriangle.js`). `u_ModelMatrix` – это `uniform`-переменная, в ней передается матрица вращения из программы на JavaScript:

```
7   ' gl_Position = u_ModelMatrix * a_Position;\n' +
```

Переменная `ANGLE_STEP` (строка 17) определяет скорость поворота в градусах в секунду и устанавливается в значение 45:

```
17 var ANGLE_STEP = 45.0;
```

Определение функции `main()` начинается в строке 19, но так как код в строках с 19 по 37, определяющий координаты вершин, остался прежним, мы его просто опустили.

Первое из трех отличий заключается в однократном определении цвета очистки: в строке 44. Затем, в строке 47, извлекается ссылка на переменную `u_ModelMatrix` в вершинном шейдере. Так как эта ссылка не изменяется в течение всего времени выполнения программы, эту операцию эффективнее выполнять только один раз:

```
47 var u_ModelMatrix = gl.getUniformLocation(gl.program, 'u_ModelMatrix');
```

Переменная `u_ModelMatrix` используется в функции `draw()` (в строке 99), которая рисует треугольник.

Изначально переменной `currentAngle` присваивается значение 0 (градусов) — она хранит угол, на какой следует повернуть треугольник относительно исходного положения в следующем цикле рисования. Она используется для вычисления матрицы вращения. Переменная `modelMatrix`, что определяется в строке 56, — это объект `Matrix4`, используемый для сохранения матрицы вращения внутри функции `draw()`. Эту матрицу можно было бы создавать в функции `draw()`; однако это потребовало бы создавать новый объект `Matrix4` при каждом вызове `draw()`, что довольно неэффективно. По этой причине объект создается в строке 56 и затем передается в вызов `draw()` в строке 61.

В строках с 59 по 64 находится реализация механизма 1, в виде функции `tick`, которая вызывается снова и снова для рисования треугольника. Прежде чем переходить к знакомству с механизмом, который снова и снова вызывает эту функцию, давайте посмотрим, что происходит внутри нее:

```
53 // Текущий угол поворота треугольника
54 var currentAngle = 0.0;
55 // Объект Matrix4 для модели преобразования
56 var modelMatrix = new Matrix4();
57
58 // Начать рисование треугольника <- (2)
59 var tick = function() {
60     currentAngle = animate(currentAngle); // Изменить угол поворота
61     draw(gl, n, currentAngle, modelMatrix, u_ModelMatrix);
62     requestAnimationFrame(tick); // Потребовать от браузера вызвать tick
63 };
64 tick();
```

Внутри функции `tick()` вызывается функция `animate()` (строка 60), которая изменяет текущий угол поворота треугольника, а вслед за ней вызывается `draw()` (строка 61), которая рисует треугольник с помощью `gl.drawArrays()`.

Функция `draw()` получает матрицу вращения, поворачивающую треугольник на `currentAngle` градусов, и передает ее дальше, вершинному шейдеру, через переменную `u_ModelMatrix`, после чего вызывает `gl.drawArrays()` (строки со 104 по 110). Этот код выглядит немного сложным, поэтому давайте исследуем каждую его часть в отдельности.

Повторяющиеся вызовы функции рисования (tick())

Как описывалось выше, чтобы воссоздать эффект вращения треугольника, необходимо выполнить следующие два повторяющихся шага: (1) изменить текущий угол поворота треугольника (переменная `currentAngle`) и (2) вызвать функцию рисования, передав ей угол поворота. Строки с 59 по 64 реализуют эти операции.

В данном примере эти задачи решаются в три действия (строки 60, 61 и 62), сгруппированные в одну анонимную функцию, которая присвоена переменной `tick` (см. рис. 4.8). Анонимная функция здесь используется, чтобы обеспечить передачу локальных переменных, объявленных в `main()` (`gl`, `n`, `currentAngle`, `modelMatrix` и `u_ModelMatrix`) функции `draw()` (строка 61). Желаящие освежить информацию об анонимных функциях могут обратиться к главе 2 «Первые шаги в WebGL», где анонимная функция используется для регистрации обработчика события.



Рис. 4.8. Действия, выполняемые в функции `tick`

Этот подход можно использовать для воспроизведения всех типов анимаций. Это – один из главных приемов в воспроизведении трехмерной графики.

Вызов `requestAnimationFrame()` в строке 62, требует от браузера, чтобы он вызвал функцию, указанную в первом параметре некогда в будущем, и тогда вновь будут выполнены все три операции, реализованные в виде функции `tick()`. С функцией `requestAnimationFrame()` мы познакомимся чуть ниже, а пока завершим исследование действий, выполняемых в `tick()`.

Рисование треугольника после поворота на указанный угол (`draw()`)

Функция `draw()` принимает следующие пять параметров:

- `gl`: контекст отображения, в котором выполняется рисование;
- `n`: число вершин;
- `currentAngle`: текущий угол поворота;
- `modelMatrix`: объект `Matrix4` для хранения матрицы вращения, вычисленной с использованием `currentAngle`;
- `u_ModelMatrix`: ссылка на `uniform`-переменную, через которую передается `modelMatrix`.

Фактический код функции находится в строках с 99 по 111:

```

99 function draw(gl,n, currentAngle, modelMatrix, u_ModelMatrix){      <-(3)
100     // Определить матрицу вращения
101     modelMatrix.setRotate(currentAngle, 0, 0, 1);
102
```



```

103 // Передать матрицу в вершинный шейдер
104 gl.uniformMatrix4fv( u_ModelMatrix, false, modelMatrix.elements);
105
106 // Очистить <canvas>
107 gl.clear(gl.COLOR_BUFFER_BIT);
108
109 // Нарисовать треугольник
110 gl.drawArrays(gl.TRIANGLES, 0, n);
111 }

```

В строке 101, вызовом метода `setRotate()` из библиотеки `cuon-matrix.js`, определяется матрица вращения и сохраняется в `modelMatrix`:

```
101 modelMatrix.setRotate(currentAngle, 0, 0, 1);
```

Далее, в строке 104, эта матрица передается в вершинный шейдер, вызовом `gl.uniformMatrix4fv()`:

```
104 gl.uniformMatrix4fv(u_ModelMatrix, false, modelMatrix.elements);
```

Потом, в строке 107, выполняется очистка области рисования `<canvas>` и в строке 110 вызывается метод `gl.drawArrays()`, который запускает вершинный шейдер, чтобы фактически нарисовать треугольник. Все эти действия совпадают с теми, что выполнялись и в предыдущих примерах.

А теперь перейдем к третьему действию, вызову функции `requestAnimationFrame()`, требующей от браузера вызвать функцию `tick()` некогда в будущем.

Запрос на повторный вызов (`requestAnimationFrame()`)

Обычно, если требуется повторно выполнить некоторую функцию (или функции), мы используем метод `setInterval()`.

`setInterval(func, delay)`

Многokrатно вызывает функцию `func` через равные интервалы времени `delay`.

Параметры	<code>func</code>	Определяет функцию для вызова.
	<code>delay</code>	Определяет интервал времени (в миллисекундах).

Возвращаемое значение	Числовой идентификатор таймера
------------------------------	--------------------------------

Однако, поскольку этот метод появился задолго до того, как браузеры стали поддерживать интерфейс с вкладками, он выполняется, независимо от того, какая вкладка в данный момент активна. Это может приводить к ухудшению общей производительности, поэтому совсем недавно появился новый метод `requestAnimationFrame()`. Вызов функции, запланированный с помощью этого

метода, производится, только когда вкладка, где была определена эта функция, активна. Так как метод `requestAnimationFrame()` является новым, он еще не стандартизован и определен в сторонней библиотеке `webgl-utils.js`, разработанной компанией Google, скрывающей различия между разными браузерами.

`requestAnimationFrame(func)`

Требует от браузера вызвать функцию `func` для перерисовывания (см. рис. 4.9). Это требование необходимо повторять после каждого вызова.

Параметры `func`

Определяет функцию для вызова. Функция должна принимать параметр `time`, в котором передается текущее время.

Возвращаемое значение Числовой идентификатор запроса



Рис. 4.9. Механизм действия `requestAnimationFrame()`

С помощью этого метода можно обеспечить автоматическое отключение воспроизведения анимационного эффекта в неактивной вкладке и тем самым уменьшить нагрузку на браузер. Обратите внимание на отсутствие возможности указать интервал, через которые должна вызываться функция – функция `func` (первый параметр) будет вызвана, когда браузер решит, что веб-страницу с указанным элементом `element` (второй параметр)¹ нужно перерисовать. Кроме того, после вызова функции необходимо вновь потребовать от браузера вызвать функцию в будущем, потому что предыдущее требование автоматически аннулируется после его выполнения. Требование, обеспечивающее повторный вызов, посылается в строке 62, когда тело функции `tick()` будет выполнено:

¹ Необязательный параметр, не используется в Firefox и IE. – Прим. перев.

```
62 requestAnimationFrame(tick); // Потребовать от браузера вызвать tick
```

Отменить требование на повторный вызов можно с помощью `cancelAnimationFrame()`.

cancelAnimationFrame(requestID)

Отменяет требование на повторный вызов функции, переданное вызовом `requestAnimationFrame()`.

Параметры requestID Значение, возвращаемое вызовом `requestAnimationFrame()`.

Возвращаемое значение нет

Изменение угла поворота (animate())

В заключение давайте посмотрим, как осуществляется изменение текущего угла поворота. Программа хранит текущий угол поворота треугольника (то есть, величину угла в градусах, на который повернут треугольник относительно исходного положения) в переменной `currentAngle` (объявлена в строке 54). Она вычисляет, каким должен быть следующий угол поворота, опираясь на текущий.

Изменение значения переменной `currentAngle` вынесено в отдельную функцию `animate()`, которая вызывается в строке 60. Определение этой функции начинается в строке 115. Она принимает единственный параметр `angle`, представляющий текущий угол поворота, и возвращает новый угол поворота:

```
60 currentAngle = animate(currentAngle); // Изменить угол поворота
61 draw(gl, n, currentAngle, modelMatrix, u_ModelMatrix);
...
113 // Момент времени, когда функция была вызвана в последний раз
114 var g_last = Date.now();
115 function animate(angle) {
116     // Вычислить прошедшее время
117     var now = Date.now();
118     var elapsed = now - g_last; // в миллисекундах
119     g_last = now;
120     // Изменить текущий угол поворота (в соответствии с прошедшим временем)
121     var newAngle = angle + (ANGLE_STEP * elapsed) / 1000.0;
122     return newAngle %= 360;
123 }
```

Алгоритм изменения текущего угла поворота довольно сложен, поэтому проще будет пояснить его суть с помощью рис. 4.10.

Рис. 4.10 иллюстрирует следующее:

- функция `tick()` вызывается в момент времени `t0`; она вызывает `draw()`, чтобы нарисовать треугольник, и вновь регистрирует `tick()` для вызова в будущем;
- функция `tick()` вызывается в момент времени `t1`; она вызывает `draw()`, чтобы нарисовать треугольник, и вновь регистрирует `tick()` для вызова в будущем;

- функция `tick()` вызывается в момент времени t_2 ; она вызывает `draw()`, чтобы нарисовать треугольник, и вновь регистрирует `tick()` для вызова в будущем.

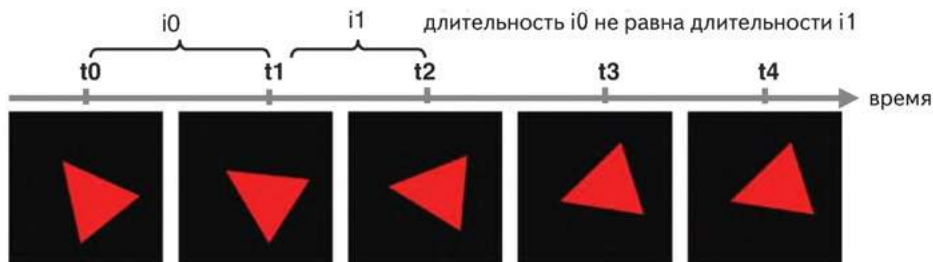


Рис. 4.10. Интервалы времени между вызовами `tick()` могут различаться

Проблема в том, что интервалы между моментами времени t_0 и t_1 , t_1 и t_2 , и между t_2 и t_3 , могут быть разными, потому что нагрузка на браузер с течением времени может изменяться. То есть, разность $t_1 - t_0$ может отличаться от разности $t_2 - t_1$.

Если интервалы времени могут быть разными, следовательно, простое увеличение текущего угла поворота на постоянную величину (градусы в секунду) в каждом вызове `tick()` приведет к скачкообразному изменению скорости вращения треугольника.

Поэтому функция `animate()` должна определять новый угол поворота, исходя из интервала времени, прошедшего с момента последнего ее вызова. С этой целью она хранит время последнего вызова в переменной `g_last` и запоминает текущее время в переменной `now`. Затем она вычисляет продолжительность интервала времени, прошедшего с момента предыдущего вызова, находя разность `now - g_last`, и сохраняет результат в переменной `elapsed` (строка 118). После этого, исходя из значения `elapsed`, в строке 121, вычисляется величина угла поворота, как показано ниже.

```
121  varnewAngle = angle + (ANGLE_STEP * elapsed) / 1000.0;
```

Обеим переменным, `g_last` и `now`, присваивается значение, возвращаемое методом `now()` объекта `Date`, измеряемое в миллисекундах (1/1000 секунды). Поэтому, чтобы вычислить угол поворота, исходя из скорости вращения `ANGLE_STEP` (градусы в секунду), достаточно просто умножить `ANGLE_STEP` на `elapsed/1000`. Фактически же, в строке 121, сначала выполняется умножение `ANGLE_STEP` на `elapsed`, а затем результат делится на 1000. Реализовано так потому, что такая последовательность операций дает чуть более точный результат, но сути это не меняет.

Наконец, в строке 122 обрабатывается выход величины угла поворота `newAngle` за границу 360 (градусов) и полученный результат возвращается вызывающей программе.

Если теперь открыть в браузере файл `RotatedTriangle.html`, вы увидите треугольник, вращающийся с постоянной скоростью. Этот подход к реализации анимации мы будем использовать в оставшихся главах, поэтому очень важно, чтобы вы поняли его суть.

Эксперименты с примером программы

Мы предлагаем реализовать в этом разделе анимационный эффект, состоящий из нескольких преобразований. Программа `RotatingTranslatedTriangle` перемещает треугольник на 0.35 единицы в положительном направлении по оси X, затем поворачивает его со скоростью 45 градусов в секунду.

Этого легко добиться, если вспомнить, что множество трансформаций можно объединить путем умножения матриц (см. главу 3).

Для этого нам нужно всего лишь добавить операцию перемещения в строку 102. Так как переменная `modelMatrix` уже хранит матрицу вращения, мы можем задействовать метод `translate()`, вместо `setTranslate()`, чтобы умножить `modelMatrix` на матрицу перемещения:

```
99 function draw(gl, n, currentAngle, modelMatrix, u_ModelMatrix) {  
100   // Определить матрицу вращения  
101   modelMatrix.setRotate(currentAngle, 0, 0, 1);  
102   modelMatrix.translate(0.35, 0, 0);  
103   // Передать матрицу в вершинный шейдер  
104   gl.uniformMatrix4fv( u_ModelMatrix, false, modelMatrix.elements);
```

Если теперь открыть пример в браузере, вы увидите анимацию, изображенную на рис. 4.11.



Рис. 4.11. Несколько наложенных друг на друга изображений, воспроизводимых программой `RotatingTranslatedTriangle`

Для тех, кому нравится иметь более полный контроль над ситуацией, на сайте книги² приводится пример программы `RotatingTriangle_withButtons`, позво-

² <https://sites.google.com/site/webglbook/>

ляющей управлять скоростью вращения с помощью кнопок (см. рис. 4.12), которые находятся под элементом `<canvas>`.

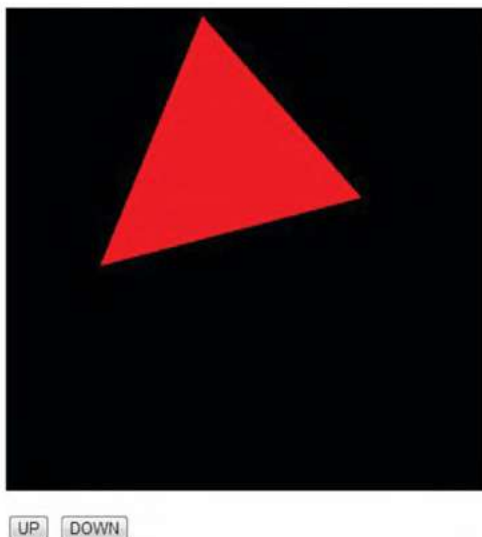


Рис. 4.12. RotatingTriangle_withButtons

В заключение

В этой главе мы исследовали приемы преобразования фигур с применением библиотеки матричных преобразований, объединяя несколько простых преобразований в более сложные, и приемы воспроизведения анимационных эффектов. Самое важное, о чем рассказывалось в этой главе: (1) сложные преобразования могут быть реализованы путем умножения последовательности простых матриц преобразований; (2) путем многократного применения преобразований можно воспроизводить анимационные эффекты.

Глава 5, «Цвет и текстура», завершает охват основных приемов воспроизведения трехмерной графики. В ней мы займемся исследованием цвета и текстуры. Овладев описываемыми там приемами, вы сможете создавать собственные WebGL-программы и будете готовы приступить к исследованию более продвинутых возможностей WebGL.



ГЛАВА 5.

Цвет и текстура

В предыдущих главах описывались ключевые понятия, составляющие основу WebGL, на примерах программ, реализующих управление двухмерными фигурами. В результате у вас должно было сложиться полное представление о том, как создавать одноцветные геометрические фигуры с использованием возможностей WebGL. Опираясь на эти основы, вы теперь можете еще больше углубиться в WebGL и заняться исследованием следующих трех тем:

- передача других данных, таких как цвет, в вершинный шейдер;
- преобразование фигур в фрагменты, которое имеет место между вершинным и фрагментным шейдерами, и известно как процесс растеризации;
- отображение изображений (или текстур) на поверхность фигуры или объекта.

Это – последняя глава, в которой рассматриваются ключевые функциональные возможности WebGL. После прочтения ее вы будете полностью понимать приемы и механизмы управления цветом и текстурами, и достигнете такого уровня владения WebGL, который позволит вам самостоятельно создавать сложные трехмерные сцены.

Передача другой информации в вершинные шейдеры

В предыдущих примерах программ мы сначала создавали единственный буферный объект, сохраняли в нем координаты вершин и затем передавали в вершинный шейдер. Однако, помимо координат, в вершинный шейдер требуется передавать и другую информацию, такую как цвет или размер точки. В качестве примера рассмотрим программу, созданную в главе 3, «Рисование и преобразование треугольников», рисующую три точки: `MultiPoint.js`. В шейдер, помимо координат вершин, мы дополнительно передавали размеры точек. Правда размеры всех точек были одинаковыми и фиксированными, и определялись внутри шейдера, а не во внешней программе:

```
3 var VSHADER_SOURCE =  
4   'attribute vec4 a_Position;\n' +
```

```
5  'void main() {\n' +  
6  '    gl_Position = a_Position;\n' +  
7  '    gl_PointSize = 10.0;\n' +  
8  '}\n'
```

В строке 6 в переменную `gl_Position` записываются координаты вершины, а в строке 7 – в переменную `gl_PointSize` – фиксированный размер точки, равный 10.0. Если теперь у вас вдруг появится желание организовать управление размерами точек из программы на JavaScript, вам потребуется предусмотреть передачу размера вместе с координатами.

Рассмотрим пример программы `MultiAttributeSize`, цель которой – нарисовать три точки разных размеров: 10.0, 20.0 и 30.0, соответственно (см. рис. 5.1).

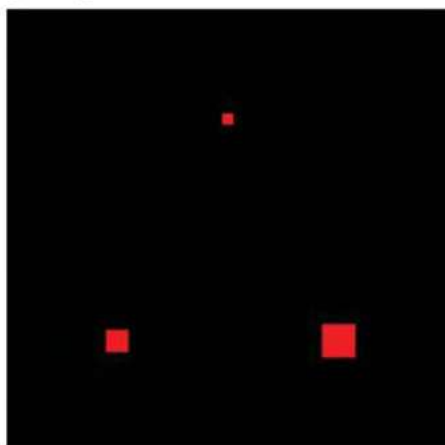


Рис. 5.1. `MultiAttributeSize`

В предыдущей главе мы реализовали следующие шаги по передаче координат вершин:

1. Создать буферный объект.
2. Указать тип буферного объекта.
3. Записать координаты в буферный объект.
4. Присвоить ссылку на буферный объект переменной-атрибуту.
5. Разрешить присваивание.

Если теперь потребуется передать в вершинный шейдер дополнительную информацию, достаточно будет просто выполнить эти шаги для каждого типа такой информации. А теперь рассмотрим пример программы, использующей несколько буферных объектов.

Пример программы (*MultiAttributeSize.js*)

В листинге 5.1 приводится содержимое файла `MultiAttributeSize.js`. Фрагментный шейдер в ней остался тем же, как в программе `MultiPoint.js`, поэтому

мы его опустили. Вершинный шейдер претерпел небольшие изменения – в нем появилась новая переменная-атрибут, определяющая размер точки. Числовые метки от 1 до 5 справа в листинге отмечают пять шагов, перечисленных выше.

Листинг 5.1. MultiAttributeSize.js

```

1 // MultiAttributeSize.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'attribute float a_PointSize;\n' +
6   'void main() {\n' +
7     '  gl_Position = a_Position;\n' +
8     '  gl_PointSize = a_PointSize;\n' +
9   '}\n';
10 ...
17 function main() {
18   ...
34   // Определить координаты вершин
35   var n = initVertexBuffers(gl);
36   ...
47   // Нарисовать три точки
48   gl.drawArrays(gl.POINTS, 0, n);
49 }
50
51 function initVertexBuffers(gl) {
52   var vertices = new Float32Array([
53     0.0, 0.5, -0.5, -0.5, 0.5, -0.5
54   ]);
55   var n = 3;
56
57   var sizes = new Float32Array([
58     10.0, 20.0, 30.0 // Размеры точек
59   ]);
60
61   // Создать буферные объекты
62   var vertexBuffer = gl.createBuffer();                <-(1)
63   var sizeBuffer = gl.createBuffer();                  <-(1')
64   ...
69   // Записать координаты вершин в буферный объект и разрешить присваивание
70   gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);        <-(2)
71   gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW); <-(3)
72   var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
73   ...
77   gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0); <-(4)
78   gl.enableVertexAttribArray(a_Position); <-(5)
79
80   // Записать размеры точек в буферный объект и разрешить присваивание
81   gl.bindBuffer(gl.ARRAY_BUFFER, sizeBuffer);          <-(2')
82   gl.bufferData(gl.ARRAY_BUFFER, sizes, gl.STATIC_DRAW); <-(3')
83   var a_PointSize = gl.getAttribLocation(gl.program, 'a_PointSize');
84   ...
88   gl.vertexAttribPointer(a_PointSize, 1, gl.FLOAT, false, 0, 0); <-(4')

```



```
89  gl.enableVertexAttribArray(a_PointSize);
    ...
94  return n;
95 }
```

Для начала исследуем вершинный шейдер в листинге 5.1. Как можно заметить, в него была добавлена переменная-атрибут `a_PointSize`, через которую из программы на JavaScript передаются размеры точек. Эта переменная объявлена в строке 5 с типом `float`, а в строке 8 ее значение присваивается переменной `gl_PointSize`. Больше никаких изменений в вершинном шейдере нет, но нам еще нужно внести изменения в функцию `initVertexBuffers()` и реализовать в ней поддержку нескольких буферных объектов. Давайте рассмотрим эту функцию поближе.

Создание нескольких буферных объектов

Определение функции `initVertexBuffers()` начинается в строке 51, а в строках с 52 по 54 определяются координаты вершин. Далее, в строке 57, определяется массив `sizes` с размерами точек:

```
57  var sizes = new Float32Array([
58      10.0, 20.0, 30.0 // Размеры точек
59  ]);
```

В строке 62 создается буферный объект, в котором будут храниться координаты вершин, а в строке 63 – еще один буферный объект (`sizeBuffer`) для хранения массива «размеров точек».

В строках с 70 по 78 программа связывает буферный объект, хранящий координаты, записывает в него данные и, наконец, присваивает его переменной-атрибуту и разрешает присваивание. Это те же самые действия, что выполнялись в предыдущих примерах программ.

В строках с 80 по 89 находится новый код, обрабатывающий размеры точек. Однако для них выполняются те же шаги, что и для координат вершин. В строке 81 указывается тип буферного объекта, хранящего размеры точек (`sizeBuffer`), в строке 82 в него записываются данные, в строке 88 ссылка на буферный объект присваивается переменной-атрибуту `a_PointSize` и затем разрешается присваивание.

Выполнив все эти действия, функция `initVertexBuffers()` завершает свою работу, а система WebGL приобретает внутреннее состояние, изображенное на рис. 5.2. Как видите, было создано два разных буферных объекта и ссылки на них присвоены двум разным переменным-атрибутам.

Когда в строке 48 будет вызвана функция `gl.drawArrays()`, все данные, хранящиеся в буферных объектах, будут последовательно переданы в соответствующие переменные-атрибуты. Присвоив эти данные переменным `gl_Position` (строка 7) и `gl_PointSize` (строка 8) внутри вершинного шейдера (см. рис. 5.2), можно нарисовать объекты разных размеров и в разных позициях.

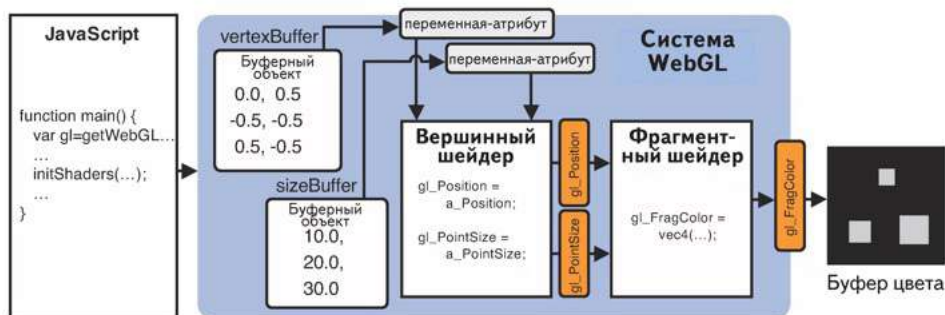


Рис. 5.2. Использование двух буферных объектов для передачи данных в вершинный шейдер

Путем создания отдельных буферных объектов и присваивания их разным переменным-атрибутам, можно передавать в вершинный шейдер разнотипную информацию о каждой вершине. Помимо координат и размеров точек, в вершинный шейдер можно также передавать цвет, координаты текстуры (подробнее об этом рассказывается далее в этой главе) и нормали (см. главу 7).

Параметры *stride* и *offset* метода `gl.vertexAttribPointer()`

Буферные объекты отлично подходят для работы с небольшими объемами данных, но нетрудно понять, насколько сложным может быть управление большими трехмерными объектами, насчитывающими тысячи вершин. Например, представьте, что потребовалось вручную проверить каждый из этих массивов, когда общее число точек в `MultiAttributeSize.js` достигает 1000.¹ Однако, WebGL позволяет хранить координаты вершин и размеры вместе, и поддерживает механизмы доступа к разнотипным данным. Например, координаты и размеры можно сгруппировать, как показано ниже (см. листинг 5.2), и такой прием часто называется **перемежением** (interleaving).

Листинг 5.2. Массив, включающий разнотипную информацию о вершинах

```
var verticesSizes = new Float32Array([
    // Координаты вершин и размеры точек
    0.0, 0.5, 10.0, // Первая точка
    -0.5, -0.5, 20.0, // Вторая точка
    0.5, -0.5, 30.0 // Третья точка
]);
```

Как только что описывалось, после сохранения разнотипной информации о вершинах в общем буферном объекте возникает потребность в механизме досту-

¹ На практике, благодаря наличию инструментов трехмерного моделирования, фактически генерирующих такие объемы данных, нет необходимости вручную вводить такие объемы или визуально проверять их целостность. Использование инструментов моделирования и данных, которые они генерируют, будет обсуждаться в главе 10.

па к разным элементам данных. Для этого можно использовать пятый (*stride*) и шестой (*offset*) аргументы метода `gl.vertexAttribPointer()`, как показано в следующем примере.

Пример программы (*MultiAttributeSize_Interleaved.js*)

Давайте напишем пример программы `MultiAttributeSize_Interleaved`, в котором в вершинный шейдер будет передаваться разнотипная информация, так же как в примере `MultiAttributeSize.js` (см. листинг 5.1), за исключением того, что данные будут храниться в общем массиве или буфере. В листинге 5.3 приводится исходный текст программы, где вершинный и фрагментный шейдеры остались прежними, как в примере `MultiAttributeSize.js`.

Листинг 5.3. `MultiAttributeSize_Interleaved.js`

```
1 // MultiAttributeSize_Interleaved.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'attribute float a_PointSize;\n' +
6   'void main() {\n' +
7     '  gl_Position = a_Position;\n' +
8     '  gl_PointSize = a_PointSize;\n' +
9   '}\n';
10 ...
17 function main() {
18   ...
34   // Определить координаты и размеры точек
35   var n = initVertexBuffers(gl);
36   ...
48   gl.drawArrays(gl.POINTS, 0, n);
49 }
50
51 function initVertexBuffers(gl) {
52   var verticesSizes = new Float32Array([
53     // Координаты и размеры точек
54     0.0, 0.5, 10.0, // Первая точка
55     -0.5, -0.5, 20.0, // Вторая точка
56     0.5, -0.5, 30.0 // Третья точка
57   ]);
58   var n = 3;
59
60   // Создать буферные объекты
61   var vertexSizeBuffer = gl.createBuffer();
62   ...
67   // Записать координаты и размеры в буферы и разрешить их
68   gl.bindBuffer(gl.ARRAY_BUFFER, vertexSizeBuffer);
69   gl.bufferData(gl.ARRAY_BUFFER, verticesSizes, gl.STATIC_DRAW);
70
71   var FSIZE = verticesSizes.BYTES_PER_ELEMENT;
```



```

72 // Получить ссылку на a_Position, выделить буфер и разрешить его
73 var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
74 ...
78 gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, FSIZE * 3, 0);
79 gl.enableVertexAttribArray(a_Position); // Разрешить использование буфера
80
81 // Получить ссылку на a_PointSize, выделить буфер и разрешить его
82 var a_PointSize = gl.getAttribLocation(gl.program, 'a_PointSize');
83 ...
87 gl.vertexAttribPointer(a_PointSize, 1, gl.FLOAT, false, FSIZE*3, FSIZE*2);
88 gl.enableVertexAttribArray(a_PointSize); // Разрешить использование буфера
89 ...
93 return n;
94 }

```

Алгоритм работы функции `main()` остался тем же, что и в примере `MultiAttributeSize.js` – на этот раз изменилась только функция `initVertexBuffers()`, поэтому посмотрим, как она действует.

Прежде всего, в строках с 52 по 57, в этой функции определяется типизированный массив, как рассказывалось в описании к листингу 5.2. Далее, в строках с 61 по 69, выполняются уже привычные операции: создается буферный объект (строка 61), указывается тип этого объекта (строка 68) и затем в него записываются данные (строка 69). Далее, в строке 71, в переменной `FSIZE` сохраняется размер (число байтов) одного элемента массива `verticeSizes`. Этот размер потребуется нам позже. Размер (число байтов) элемента типизированного массива можно получить с помощью свойства `BYTES_PER_ELEMENT`.

В строках с 73 и далее буферный объект присваивается переменной-атрибуту. В строке 73 мы получаем ссылку на переменную-атрибут `a_Position`, точно так же, как это делалось в предыдущем примере, но на этот раз в вызов метода `gl.vertexAttribPointer()` (строка 78) передаются другие аргументы, потому что буфер теперь хранит данные двух типов: координаты вершин и размеры точек.

В главе 3 вы уже видели описание метода `gl.vertexAttribPointer()`, тем не менее, приведем его еще раз, чтобы вы могли по-новому взглянуть на его параметры `stride` и `offset`.

`gl.vertexAttribPointer(location, size, type, normalized, stride, offset)`

Присваивает буферный объект типа `gl.ARRAY_BUFFER` переменной-атрибуту `location`.

Параметры	<code>location</code>	Определяет переменную-атрибут, которой будет выполнено присваивание.
	<code>size</code>	Определяет число компонентов на вершину в буферном объекте (допустимыми являются значения от 1 до 4).
	<code>type</code>	Определяет формат данных (в данном случае <code>gl.FLOAT</code>).
	<code>normalized</code>	Либо <code>true</code> , либо <code>false</code> . Указывает на необходимость нормализации невещественных данных в диапазон <code>[0, 1]</code> или <code>[-1, 1]</code> .

<code>stride</code>	Определяет длину шага (в байтах) для извлечения информации об одной вершине, то есть, число байтов между разными элементами данных.
<code>offset</code>	Определяет смещение (в байтах) от начала буферного объекта, где хранятся данные для вершин. Если данные хранятся, начиная с самого начала буфера, в этом параметре следует передать значение 0.

Параметр `stride` определяет число байтов, занимаемых данными для одной вершины (в данном примере такими данными являются координаты и размеры точек) в буферном объекте.

В предыдущих примерах, где в буфере сохранялась лишь однотипная информация (координаты), мы передавали в параметре `stride` значение 0. Однако в этом примере в одном и том же буфере хранятся и координаты, и размеры точек, как показано на рис. 5.3.

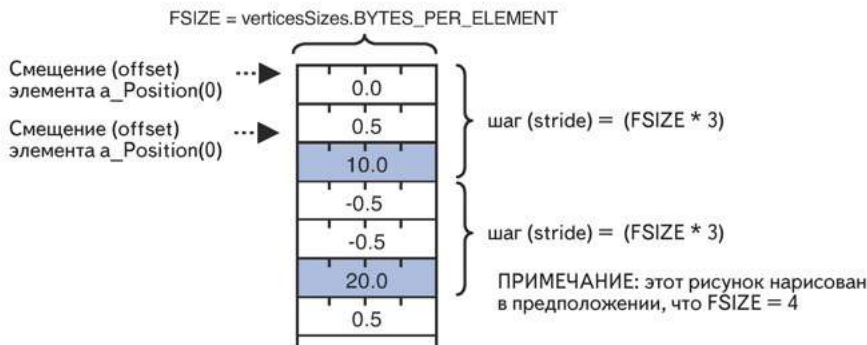


Рис. 5.3. Параметры `stride` и `offset`

Как показано на рис. 5.3, внутри каждой группы данных имеется три элемента (две координаты и один размер), поэтому размер шага (параметр `stride`) нужно установить равным трем размерам одного элемента в группе (то есть, $3 * \text{FSIZE}$ [число байтов в одном элементе массива `Floats32Array`]).

Параметр `offset` определяет расстояние (смещение) до первого элемента, который должен использоваться в этом вызове. Так как координаты вершин находятся в начале массива `verticesSizes`, смещение равно 0. В строке 78 мы указываем шаг и смещение в пятом (`stride`) и в шестом (`offset`) аргументах метода `gl.vertexAttribPointer()`:

```
78 gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, FSIZE * 3, 0);
79 gl.enableVertexAttribArray(a_Position); // Разрешить использование буфера
```

Наконец, после записи координат вершины, программа разрешает использование переменной `a_Position` в строке 79.

Далее, то же самое делается для передачи размера точки, и мы начинаем с получения ссылки на переменную `a_PointSize` в строке 82. При этом мы используем

тот же самый буфер, что перед этим использовался для передачи координат вершины. Чтобы сделать это возможным, мы задействуем шестой аргумент `offset`, в котором указываем смещение интересующих нас данных в буфере (в данном случае – размер точки) для записи в `a_PointSize`. Первые два элемента массива – это координаты вершины, соответственно смещение должно быть равно `FSIZE * 2` (см. рис. 5.3). В строке 87 можно видеть правильные значения аргументов `stride` и `offset`:

```
87 gl.vertexAttribPointer(a_PointSize, 1, gl.FLOAT, false, FSIZE * 3, FSIZE * 2);
88 gl.enableVertexAttribArray(a_PointSize); // Разрешить использование
```

В строке 88 разрешается присваивание переменной `a_PointSize` и после этого остается только нарисовать фигуру вызовом `gl.drawArrays()`.

Каждый раз, когда вызывается вершинный шейдер, WebGL извлекает данные из буферного объекта, используя значения параметров `stride` и `offset`, и переписывает их в переменные-атрибуты, используемые при рисовании (см. рис. 5.4).

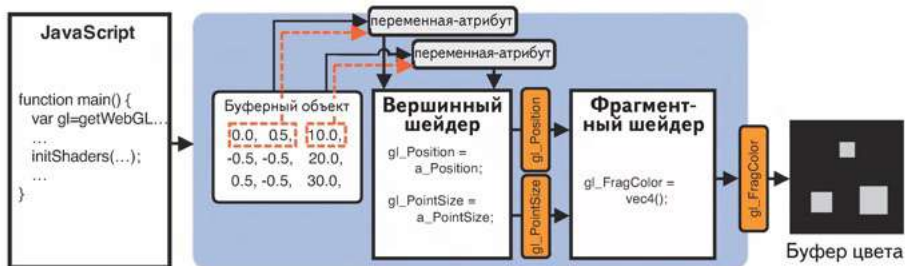


Рис. 5.4. Использование параметров `stride` и `offset` в системе WebGL

Изменение цвета (*varying-переменные*)

Теперь, когда вы знаете, как передать в вершинный шейдер разнотипную информацию, попробуем реализовать изменение цвета каждой точки. Эту задачу можно решить, используя процедуру, описанную выше, передавая вместо размеров точек их цвета. После сохранения координат вершины и цвета в буферном объекте, можно присвоить цвет переменной-атрибуту, обслуживающей цвет.

Давайте напишем такую программу и дадим ей имя `MultiAttributeColor`. Она должна рисовать три точки – красного, синего и зеленого цвета. Результат работы программы показан на рис. 5.5. (Так как эта книга черно-белая, возможно вам трудно будет заметить разницу в цвете, поэтому откройте этот пример в своем браузере.)

Многие из вас наверняка помнят из главы 2, «Первые шаги в WebGL», что атрибутами, такими как цвет, управляет фрагментный шейдер. До настоящего момента мы устанавливали цвет статически внутри фрагментного шейдера и не выполняли никаких манипуляций с ним. Однако, хотя мы и знаем, что цвет точки можно передать в вершинный шейдер через переменную-атрибут, мы не сможем присвоить значение цвета переменной `gl_FragColor` в нем, потому что эта переменная

доступна только в фрагментном шейдере (см. раздел «Фрагментный шейдер» в главе 2). Соответственно, нам нужно найти какой-то способ взаимодействия с фрагментным шейдером, чтобы передать ему информацию о цвете, предварительно переданную в вершинный шейдер (рис. 5.6).

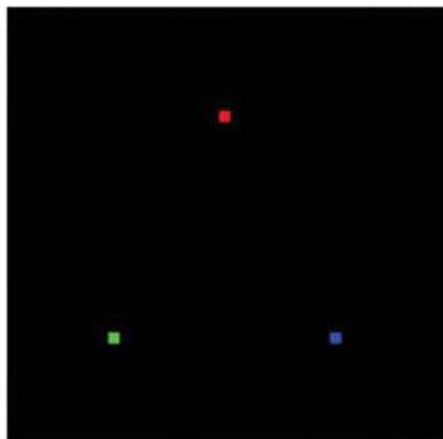


Рис. 5.5. MultiAttributeColor

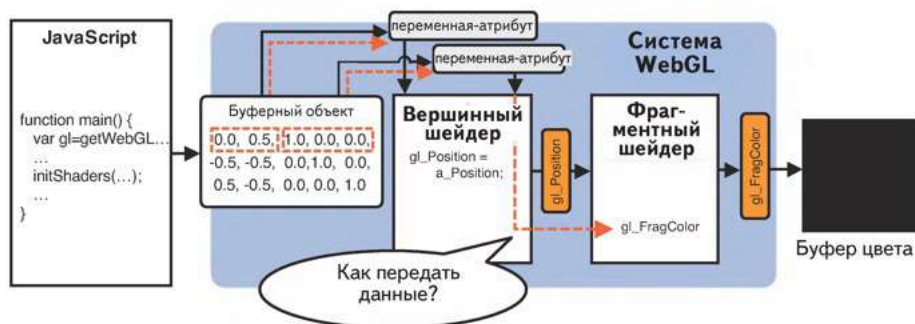


Рис. 5.6. Передача данных из вершинного шейдера во фрагментный шейдер

В примере ColoredPoints (глава 2), для передачи информации о цвете во фрагментный шейдер использовалась *uniform*-переменная; однако, из-за того, что значение переменной является «общим для всех» (то есть, не изменяется), ее нельзя использовать для передачи разных цветов. Здесь необходимо использовать новый для нас способ передачи данных во фрагментный шейдер – посредством **varying-переменной**, представляющей механизм передачи между вершинным и фрагментным шейдерами. Рассмотрим конкретный пример программы.

Пример программы (MultiAttributeColor.js)

В листинге 5.4 приводится программа, которая выглядит похожей на программу MultiAttributeSize_Stride.js из предыдущего раздела, только в ней немного изменились части, имеющие отношение к вершинному и фрагментному шейдерам.

Листинг 5.4. MultiAttributeColor.js

```

1 // MultiAttributeColor.js
2 // Vertex shader program
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'attribute vec4 a_Color;\n' +
6   'varying vec4 v_Color;\n' + // varying-переменная
7   'void main() {\n' +
8   '  gl_Position = a_Position;\n' +
9   '  gl_PointSize = 10.0;\n' +
10  '  v_Color = a_Color;\n' + // Передача данных во фрагментный шейдер
11  '}\n';
12
13 // Фрагментный шейдер
14 var FSHADER_SOURCE =
15   ...
16   'varying vec4 v_Color;\n' +
17   'void main() {\n' +
18   '  gl_FragColor = v_Color;\n' + // Принять данные из вершинного шейдера
19   '}\n';
20
21
22
23 function main() {
24   ...
25   // Определить координаты вершины и цвет
26   var n = initVertexBuffers(gl);
27   ...
28   gl.drawArrays(gl.POINTS, 0, n);
29 }
30
31
32 function initVertexBuffers(gl) {
33   var verticesColors = new Float32Array([
34     // Координаты и цвет вершин
35     0.0, 0.5, 1.0, 0.0, 0.0,
36     -0.5, -0.5, 0.0, 1.0, 0.0,
37     0.5, -0.5, 0.0, 0.0, 1.0,
38   ]);
39   var n = 3; // число вершин
40
41   // Создать буферный объект
42   var vertexColorBuffer = gl.createBuffer();
43   ...
44   // Записать координаты и цвет в буферный объект
45   gl.bindBuffer(gl.ARRAY_BUFFER, vertexColorBuffer);
46   gl.bufferData(gl.ARRAY_BUFFER, verticesColors, gl.STATIC_DRAW);
47
48   var FSIZE = verticesColors.BYTES_PER_ELEMENT;
49   // Получить ссылку на a_Position, выделить буфер и разрешить его
50   var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
51   ...
52   gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, FSIZE*5, 0);
53   gl.enableVertexAttribArray(a_Position); // Разрешить использование буфера
54
55   // Получить ссылку на a_Color, выделить буфер и разрешить его

```



```

88  var a_Color = gl.getAttribLocation(gl.program, 'a_Color');
...
93  gl.vertexAttribPointer(a_Color, 3, gl.FLOAT, false, FSIZE*5, FSIZE*2);
94  gl.enableVertexAttribArray(a_Color); // Разрешить использование буфера
...
96  return n;
97 }

```

В строке 5, в реализации вершинного шейдера, объявляется переменная-атрибут `a_Color` для приема значения цвета. Далее, в строке 6, объявляется новая `varying`-переменная `v_Color`, которая будет использоваться для передачи цвета фрагментному шейдеру. Обратите внимание, что в объявлениях `varying`-переменных можно использовать только вещественные типы (и связанные с ними типы `vec2`, `vec3`, `vec4`, `mat2`, `mat3` и `mat4`):

```

5  'attribute vec4 a_Color;\n' +
6  'varying vec4 v_Color;\n' +

```

В строке 10 значение переменной `a_Color` присваивается переменной `v_Color`, объявленной в строке 6:

```

10  '  v_Color = a_Color;\n' +

```

Но, как фрагментный шейдер получит присвоенные данные? Ответ на этот вопрос прост: нужно объявить ту же самую переменную во фрагментном шейдере, с тем же именем и того же типа, как в вершинном шейдере:

```

18  'varying vec4 v_Color;\n' +

```

В WebGL, когда `varying`-переменные, объявленные во фрагментном шейдере, своими именами и типами совпадают с переменными, объявленными в вершинном шейдере, значения, присваиваемые в вершинном шейдере автоматически передаются во фрагментный шейдер (см. рис. 5.7).

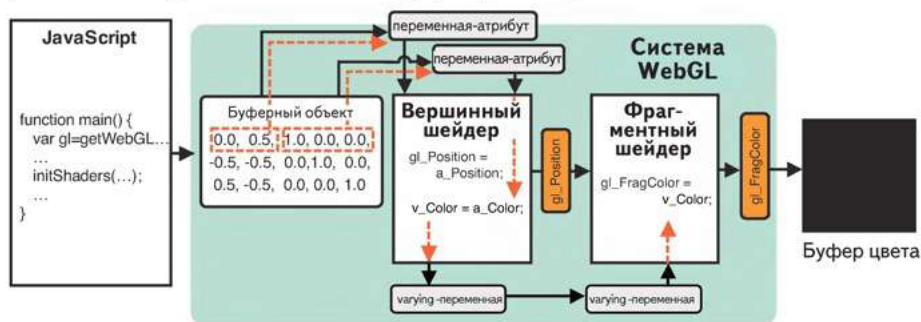


Рис. 5.7. Поведение `varying`-переменной

Итак, мы можем посылать значения во фрагментный шейдер из вершинного простым присваиванием значения `varying`-переменной (строка 10), поэтому нам остается только присвоить переменной `gl_FragColor` значение `v_Color` (стро-

ка 20). Так как `gl_FragColor` определяет цвет фрагмента, каждая точка получит свой цвет:

```
20   '   gl_FragColor = v_Color;\n' +
```

Остальной код мало чем отличается от кода программы `MultiAttributeSize.js`. Единственное отличие заключено в имени `verticesColors` типизированного массива, предназначенного для хранения информации о вершинах, объявленного в строке 58, а так же в добавлении информации о цвете, такой как `(1.0, 0.0, 0.0)`, в строке 60.

Как рассказывалось в главе 2, цвет определяется в формате RGBA, каждый компонент в котором изменяется в диапазоне 0.0–1.0. Так же как в `MultiAttributeSize_Stride.js`, мы сохраняем несколько разнотипных данных в одном массиве. Пятый (`stride`) и шестой (`offset`) аргументы в вызовах `gl.vertexAttribPointer()`, что выполняются в строках 84 и 93, изменились, из-за изменения содержимого массива `verticesColors`, в котором, вследствие введения дополнительной информации о цвете, изменился размер шага, ставший теперь равным `FSIZE * 5`.

Наконец, команда рисования в строке 54 приводит к появлению в окне браузера красной, синей и зеленой точек.

Эксперименты с примером программы

Давайте изменим первый аргумент в вызове `gl.drawArrays()`, в строке 54, на `gl.TRIANGLES` и посмотрим, что из этого получится. Желающие могут просто загрузить пример программы `ColoredTriangle` с веб-сайта книги:

```
54   gl.drawArrays(gl.TRIANGLES, 0, n);
```

Результат работы этой программы представлен на рис. 5.8. Читателям, имеющим перед глазами черно-белый рисунок, возможно, будет трудно уловить различия, но на экране вы увидите плавные переходы цветов между красным, зеленым и синим в углах.



Рис. 5.8. ColoredTriangle

Изменение значения всего одного параметра произвело потрясающий эффект – три цветные точки превратились в радужный треугольник с плавными переходами цветов. Давайте посмотрим, как это получилось.

Цветной треугольник (ColoredTriangle.js)

Мы уже занимались темой цветных треугольников в главе 3, правда там треугольники были одноцветные. В этом разделе мы объясним, как определить разные цвета для всех вершин треугольника и как обрабатывается эта ситуация WebGL, в результате чего отображаются плавные переходы цветов между вершинами.

Чтобы полностью постичь это явление, необходимо понимать, все тонкости взаимодействия вершинных и фрагментных шейдеров, а также особенности работы *varying*-переменных.

Сборка и растеризация геометрических фигур

В своих исследованиях мы будем опираться на пример программы `HelloTriangle.js` из главы 3, которая рисует простой красный треугольник. Фрагменты кода, которые нам пригодятся, показаны в листинге 5.5.

Листинг 5.5. `HelloTriangle.js` (фрагменты кода)

```
1 // HelloTriangle.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'void main() {\n' +
6   '  gl_Position = a_Position;\n' +
7   '}\n';
8
9 // Фрагментный шейдер
10 var FSHADER_SOURCE =
11   'void main() {\n' +
12   '  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);\n' +
13   '}\n';
14
15 function main() {
16   ...
17   // Определить координаты вершин
18   var n = initVertexBuffers(gl);
19   ...
20   // Нарисовать треугольник
21   gl.drawArrays(gl.TRIANGLES, 0, n);
22 }
23
24 function initVertexBuffers(gl) {
25   var vertices = new Float32Array([
26     0.0, 0.5, -0.5, -0.5, 0.5, -0.5
27   ]);
28 }
```

```

53  var n = 3; // Число вершин
    ...
74  gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);
    ...
81  return n;
82  }

```

В этой программе, после записи координат вершин в буферный объект (строки с 50 по 52 в функции `initVertexBuffers()`), буферный объект присваивается переменной-атрибуту `a_Position` в строке 74. Затем, когда `gl.drawArrays()` вызывает вершинный шейдер (строка 46), координаты трех вершин, находящиеся в буферном объекте, передаются в переменную `a_Position` (строка 4) и присваиваются переменной `gl_Position` (строка 6), что делает их доступными для фрагментного шейдера. Внутри фрагментного шейдера переменной `gl_FragColor` присваивается значение в формате RGBA (1.0, 0.0, 0.0, 1.0), обозначающее красный цвет, поэтому на экране появляется красный треугольник.

Фактически, до настоящего момента мы вообще не рассматривали суть происходящего, поэтому давайте посмотрим, как же все-таки фрагментный шейдер выполняет операции с фрагментами, когда мы передаем в `gl_Position` только координаты трех вершин.

На рис. 5.9 показано, как решается эта задача. Программа передает три вершины, но что именно указывает на то, что координаты вершин в `gl_Position` описывают вершины треугольника? Кроме того, что обеспечивает заливку треугольника одним цветом, где принимается решение о том, какие фрагменты должны быть окрашены? Наконец, когда вызывается фрагментный шейдер и как он обрабатывает каждый фрагмент?

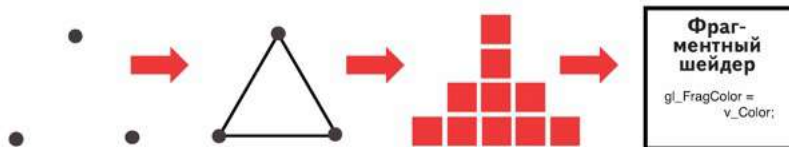


Рис. 5.9. Координаты вершин, идентификация треугольника по координатам вершин, растеризация и выполнение фрагментного шейдера

До настоящего момента мы замалчивали все эти подробности. В действительности между вершинным и фрагментным шейдерами выполняются две операции, как показано на рис. 5.10:

- **операция сборки геометрической фигуры:** на этом этапе из указанных координат вершин выполняется сборка геометрической фигуры; тип фигуры определяет первый аргумент метода `gl.drawArrays()`;
- **операция растеризации:** на этом этапе собранная геометрическая фигура преобразуется в множество фрагментов.

Как нетрудно заметить на рис. 5.10, переменная `gl_Position` фактически действует как входные данные для этапа **сборки геометрической фигуры**. Обратите внимание, что процесс сборки геометрической фигуры называется также **сборкой**

примитива, потому что простейшие фигуры, представленные в главе 2, называют также **примитивами**.



Рис. 5.10. Операции сборки и растеризации между вызовами вершинного и фрагментного шейдеров

На рис. 5.11 показаны процессы между вершинным и фрагментным шейдерами, которые фактически протекают на этапе сборки и растеризации в приложении `HelloTriangle.js`.

В листинге 5.5 методу `gl.drawArrays()` (строка 46) передается аргумент `n` со значением 3. Это означает, что вершинный шейдер будет вызван три раза.

- Шаг 1.** Вызывается вершинный шейдер и ему передаются координаты первой вершины (0.0, 0.5) из буферного объекта, полученного в переменной `a_Position`. После присваивания координат переменной `gl_Position`, они передаются дальше, процедуре сборки геометрической фигуры и сохраняются там. Как вы наверняка помните, в переменной `a_Position` передаются только две координаты, X и Y, а параметры `z` и `w` получают значения по умолчанию, поэтому фактически сохраняются координаты (0.0, 0.5, 0.0, 1.0).
- Шаг 2.** Вновь вызывается вершинный шейдер и процедуре сборки геометрической фигуры передаются координаты второй точки (−0.5, −0.5).
- Шаг 3.** Вершинный шейдер вызывается в третий раз и процедуре сборки геометрической фигуры передаются координаты третьей точки (0.5, −0.5). На этом работа вершинного шейдера заканчивается, и процедуре сборки геометрической фигуры становятся доступны все три координаты.
- Шаг 4.** Запускается процесс сборки геометрической фигуры. На основе полученных координат трех вершин и значения первого аргумента (`gl.TRIANGLES`) в вызове `gl.drawArrays()`, на этой стадии определяется, какой примитив требуется собрать. В данном случае из трех вершин собирается треугольник.
- Шаг 5.** Так как отображаемый на экране треугольник состоит из фрагментов (пикселей), геометрическая фигура преобразуется в массив фрагментов. Этот процесс называется **растеризацией**. На этом этапе формируется массив фрагментов, составляющих треугольник. Пример такого сгенерированного массива фрагментов можно видеть в рамке «Растеризация» на рис. 5.11.

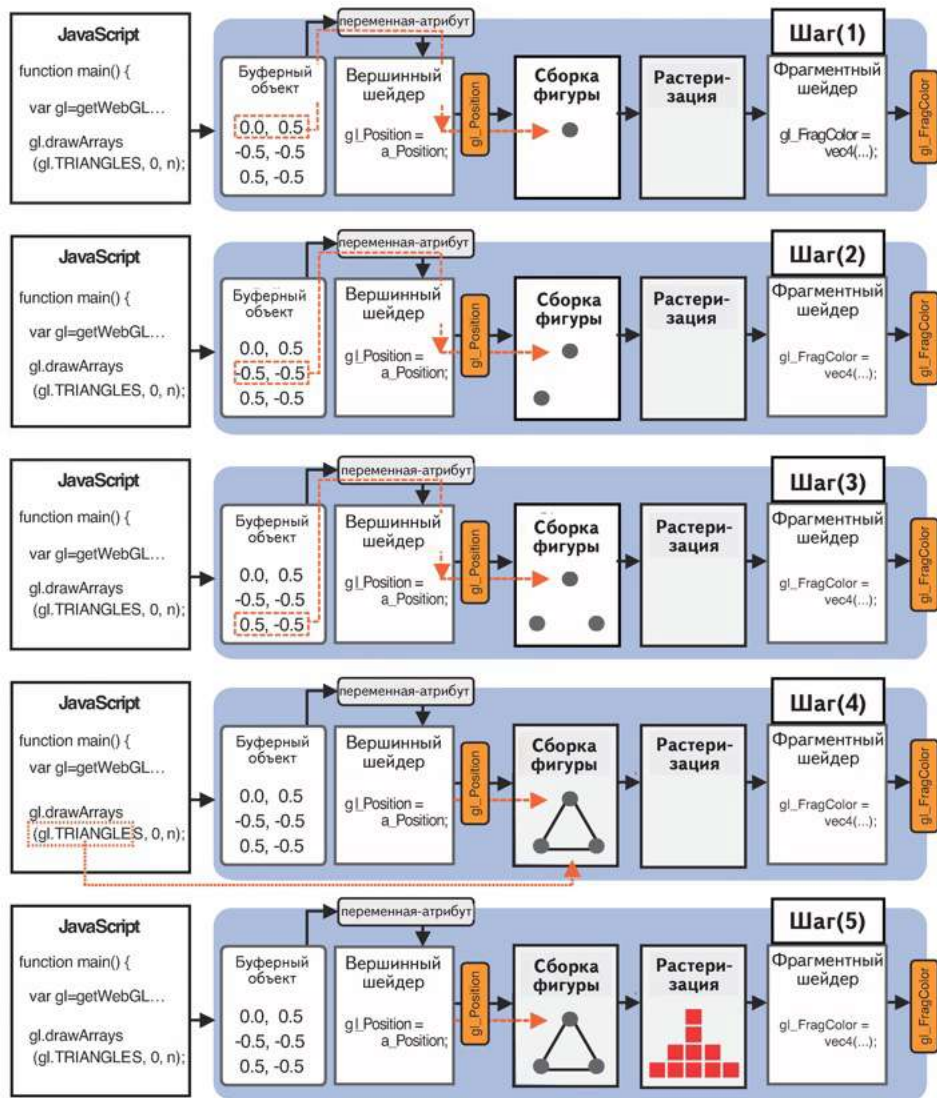


Рис. 5.11. Порядок сборки геометрической фигуры и растеризации

Хотя на этом рисунке показано всего 10 фрагментов, фактическое их число определяется площадью отображаемого треугольника.

Если в первом аргументе методу `gl.drawArrays()` передать другое значение, геометрическая фигура, собираемая на шаге 4, изменится соответственно, а также число фрагментов и их позиции, определяемые на шаге 5. Например, если в первом аргументе передать `gl.LINES`, между первыми двумя вершинами будет собрана прямая линия, а третья вершина будет отброшена. Если в первом аргу-

менте передать `gl.LINE_LOOP`, будет сгенерирована замкнутая ломаная и будет нарисован тот же треугольник, только прозрачный (не покрашенный цветом заливки).

Вызовы фрагментного шейдера

По завершении этапа растеризации, для обработки каждого созданного фрагмента вызывается фрагментный шейдер. Так, в данном примере фрагментный шейдер будет вызван 10 раз, как показано на рис. 5.12. Чтобы избежать путаницы на рисунке, мы опустили некоторые промежуточные операции. Все фрагменты передаются фрагментному шейдеру по одному, друг за другом, и для каждого фрагмента шейдер устанавливает цвет и записывает его в буфер цвета. Когда фрагментный шейдер завершит обработку последнего фрагмента на шаге 15, в окне браузера появится получившееся изображение.

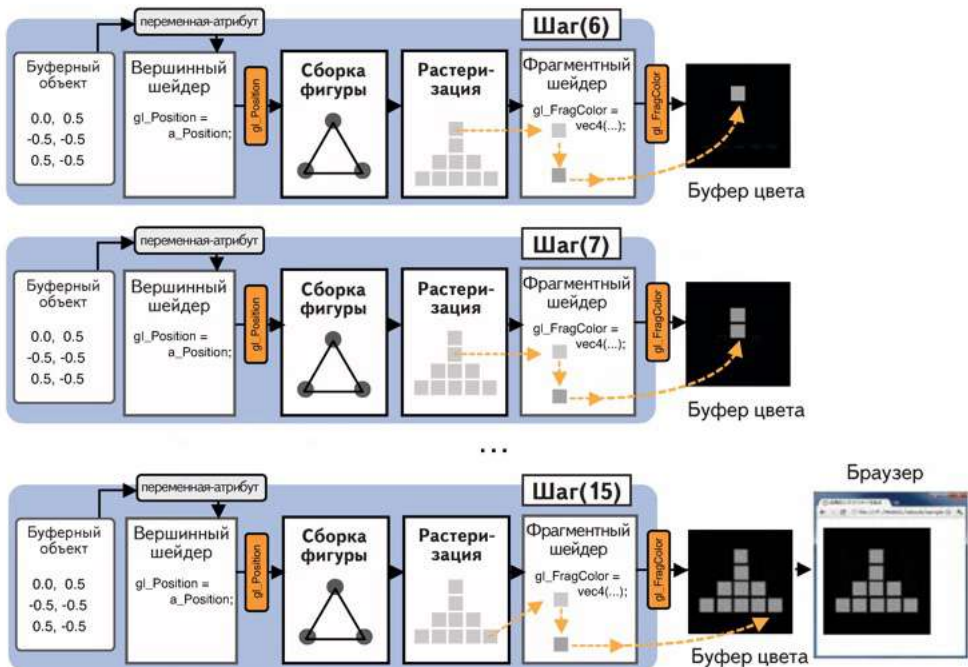


Рис. 5.12. Вызовы фрагментного шейдера

Следующий ниже фрагментный шейдер, из программы `HelloTriangle.js`, окрашивает каждый фрагмент в красный цвет. В результате в буфере цвета оказывается треугольник, залитый красным цветом, который затем отображается в окне браузера.

```
9 // Фрагментный шейдер
10 var FSHADER_SOURCE =
11     'void main() {\n' +
```



```

12   '   gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);\n' +
13   '}\n';

```

Эксперименты с примером программы

В качестве эксперимента попробуем убедиться, что фрагментный шейдер действительно вызывается для каждого фрагмента и устанавливает его цвет, исходя из местоположения этого фрагмента. Каждый фрагмент, сгенерированный на этапе растеризации, имеет определенные координаты, передаваемые фрагментному шейдеру при вызове. Эти координаты доступны через встроенные переменные (табл. 5.1).

Таблица 5.1. Встроенные переменные, доступные во фрагментном шейдере

Тип и имя переменной	Описание
vec4 gl_FragCoord	Первый и второй элементы массива – координаты фрагмента в системе координат элемента <canvas> (система координат окна).

Чтобы убедиться, что фрагментный шейдер действительно вызывается для каждого фрагмента, изменим реализацию фрагментного шейдера в программе, как показано ниже:

```

1 // HelloTriangle_FragCoord.js
...
9 // Фрагментный шейдер
10 var FSHADER_SOURCE =
11   'precision mediump float;\n' +
12   'uniform float u_Width;\n' +
13   'uniform float u_Height;\n' +
14   'void main() {\n' +
15   '   gl_FragColor = vec4(gl_FragCoord.x/u_Width, 0.0,
16     '   gl_FragCoord.y/u_Height, 1.0);\n' +
17   '}\n';

```

Как видите, красная и синяя составляющие цвета каждого фрагмента вычисляются исходя из координат этого фрагмента в элементе <canvas>. Обратите внимание, что ось Y элемента <canvas> направлена в обратном направлении, в отличие от системы координат WebGL, а так как значение составляющей цвета в WebGL может изменяться в диапазоне от 0.0 до 1.0, чтобы получить значение цвета, мы делим координату на соответствующий ей размер элемента <canvas> (то есть, на 400 пикселей). Кроме того, как следует из листинга выше, во фрагментный шейдер передаются также ширина и высота в виде uniform-переменных u_Width и u_Height, которые определяются в программе с помощью методов gl.drawingBufferWidth и gl.drawingBufferHeight. Результат работы этого примера представлен на рис. 5.13, где изображен треугольник, цвет фрагментов в котором определяется программой, исходя из их местоположения. Запустив программу HelloTriangle_FragCoord, вы увидите плавный переход цвета в направлении слева сверху вправо вниз.

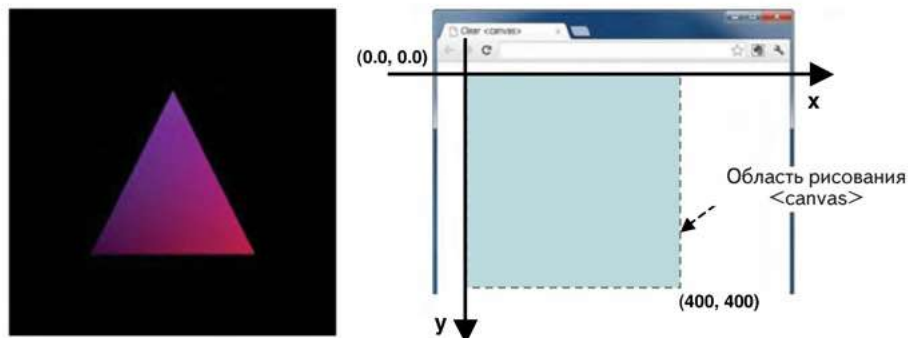


Рис. 5.13. Изменение цвета для каждого фрагмента (справа изображена система координат элемента `<canvas>`)

Так как цвет каждого фрагмента вычисляется на основе его координат, на полученном изображении можно видеть плавное изменение цвета. И снова, если на черно-белой иллюстрации в книге не видно плавного перехода цвета, запустите пример на веб-сайте книги.

Принцип действия *varying*-переменных и процесс интерполяции

На данном этапе у вас должно сложиться достаточно полное представление о происходящем между вызовами вершинного и фрагментного шейдеров (то есть, о процессах сборки геометрической фигуры и растеризации). Также вы убедились, что фрагментный шейдер вызывается для каждого фрагмента.

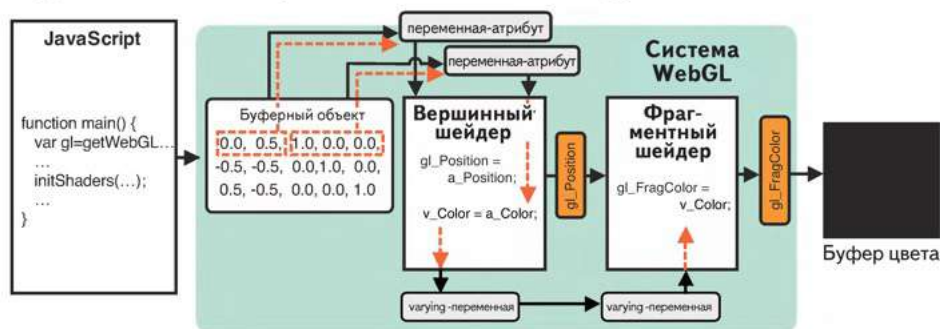


Рис. 5.14. Принцип действия *varying*-переменной (повтор рис. 5.7)

Вернемся к рис. 5.8 – к первой версии программы `ColoredTriangle` – и применим полученные знания, чтобы понять, почему получается такой интересный радужный треугольник, когда для вершин указываются разные цвета. Ранее мы видели, что значение *varying*-переменной, присвоенное внутри вершинного шейдера, передается в одноименную *varying*-переменную (того же типа) во фрагментный шейдер (см. рис. 5.14). Однако, если быть более точными, значение, при-

своеенное `varying`-переменной в вершинном шейдере, интерполируется на этапе растеризации. Соответственно, значения, фактически передаваемые фрагментному шейдеру, отличаются для разных фрагментов (см. рис. 5.15). Это объясняет суть спецификатора `varying` в объявлениях `varying`-переменных.²

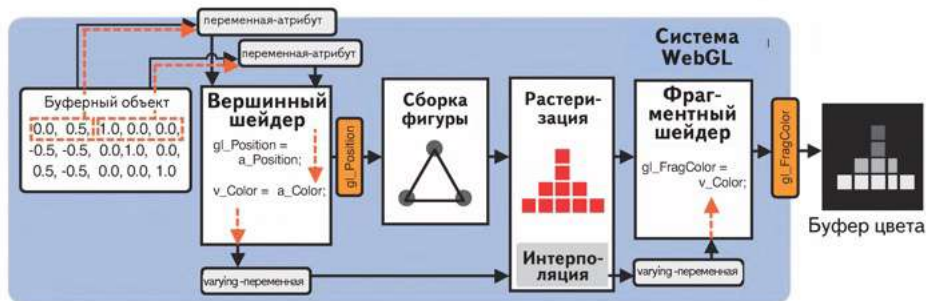


Рис. 5.15. Интерполяция значения `varying`-переменной

В частности, в приложении `ColoredTriangle`, из-за того, что мы присваиваем разные значения `varying`-переменной для каждой из трех вершин, каждый фрагмент, расположенный между вершинами должен получить свой цвет, интерполированный системой WebGL.

Например, рассмотрим случай, когда две конечные точки отрезка определены с разными цветами. Одна из вершин имеет красный цвет (1.0, 0.0, 0.0), а другая – синий (0.0, 0.0, 1.0). После того, как цвета (красный и синий) будут присвоены переменной `v_Color` в вершинном шейдере, значения RGB для каждого фрагмента, находящегося между двумя вершинами будут вычисляться и передаваться фрагментному шейдеру в его переменной `v_Color` (см. рис. 5.16).



Рис. 5.16. Интерполяция значений цвета

В данном случае составляющая R (red – красная) уменьшается от 1.0 до 0.0, составляющая B (blue – синяя) увеличивается от 0.0 до 1.0, и все значения RGB между этими двумя вершинами вычисляются соответственно позициям фрагментов. Это и называется **процессом интерполяции**. После вычисления цвета для очередного фрагмента он передается фрагментному шейдеру через переменную `v_Color`.

² `varying` – изменчивый, меняющийся. – Прим. перев.

Идентичным образом работает и программа в листинге 5.6, которая воспроизводит цветной треугольник. После того, как трем вершинам будут присвоены цвета через *varying*-переменную *v_Color* (строка 9), интерполированный цвет для каждого фрагмента передается фрагментному шейдеру в переменной *v_Color*, где присваивается переменной *gl_FragColor* в строке 19. В результате отображается цветной треугольник, как показано на рис. 5.8. Процедура интерполяции выполняется для каждой *varying*-переменной. Желаящим разобраться во всех тонкостях этого процесса мы рекомендуем обратиться к книге «Computer Graphics».

Листинг 5.6. ColoredTriangle.js

```
1 // ColoredTriangle.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE = '\
...
6   varying vec4 v_Color;\
7   void main() {\
8       gl_Position = a_Position;\
9       v_Color = a_Color;\ <- Цвет присваивается переменной v_Color в строке 59
10  }';
11
12 // Фрагментный шейдер
13 var FSHADER_SOURCE =
...
17   varying vec4 v_Color;\ <- Интерполированный цвет передается в v_Color
18   void main() {\
19       gl_FragColor = v_Color;\ <- Цвет присваивается переменной gl_FragColor
20   }';
21
22 function main() {
...
53   gl.drawArrays(gl.TRIANGLES, 0, n);
54 }
55
56 function initVertexBuffers(gl) {
57   var verticesColors = new Float32Array([
58     // Координаты вершин и цвета
59     0.0, 0.5, 1.0, 0.0, 0.0,
60     -0.5, -0.5, 0.0, 1.0, 0.0,
61     0.5, -0.5, 0.0, 0.0, 1.0,
62   ]);
...
99 }
```

Итак, в этом разделе мы подчеркнули особую важность процедуры растеризации, которая выполняется между вершинным и фрагментным шейдерами. Растеризация занимает ключевое положение в трехмерной графике, она получает на входе геометрическую фигуру и формирует массив фрагментов, составляющих эту фигуру. После преобразования геометрической фигуры в массив фрагментов (растеризации), каждому фрагменту можно придать свой цвет внутри фрагментного шейдера. Этот цвет может быть интерполирован или установлен непосредственно программистом.

Наложение изображения на прямоугольник

В предыдущем разделе мы выяснили, как управлять цветом при рисовании фигур и как механизм интерполяции позволяет получить плавные переходы цвета. Однако, несмотря на большие возможности, этот подход оказывается недостаточно мощным, когда дело доходит до воспроизведения сложных сцен. Например, вас ждут большие проблемы, если вы попытаетесь воспроизвести изображение каменной стены, как показано на рис. 5.17, так как вам потребуется нарисовать множество треугольников и для каждого из них определить цвет и координаты вершин.

Как вы наверняка догадались, решение этой проблемы является одним из важнейших приемов в технологии отображения трехмерной графики. Этот прием называется **наложением текстуры** и позволяет воссоздавать реалистичные изображения разных материалов. В действительности этот прием достаточно прост и заключается в наложении изображения (подобно переводной картинке) на поверхность геометрической фигуры. Накладывая изображение на прямоугольник, составленный из двух треугольников, можно придать прямоугольнику внешний вид, подобный изображенному на рис. 5.17. Накладываемое изображение называется **текстурным изображением**, или **текстурой**.



Рис. 5.17. Пример сложной поверхности, изображающей каменную стену

Процедура наложения текстуры заключается в присваивании цветов пикселей изображения фрагментам, сгенерированным процедурой растеризации, описанной в предыдущем разделе. Пиксели, составляющие текстуру, называют **текселями** (элементами текстуры), и каждый тексель хранит информацию о своем цвете в формате RGB или RGBA (см. рис. 5.18).



Рис. 5.18. Тексели

Наложение текстуры в WebGL осуществляется в четыре этапа:

1. Подготовка изображения для наложения на геометрическую фигуру.

2. Определение метода наложения.
3. Загрузка изображения текстуры и настройка его для использования в WebGL.
4. Извлечение текстелей из изображения во фрагментном шейдере и установка цвета соответствующих фрагментов.

Чтобы понять, как действуют механизмы, вовлеченные в процесс наложения текстуры, исследуем пример программы `TextureQuad`, которая «натягивает» изображение на прямоугольник. Если запустить ее с сайта книги, вы увидите изображение, показанное на рис. 5.19 (слева).

Примечание. Если вам понадобится запускать в браузере Chrome примеры программ, использующие изображения, хранящиеся на локальном диске, добавьте параметр командной строки `--allow-file-access-from-files` в команду запуска Chrome. Это обусловлено требованиями безопасности, так как по умолчанию Chrome запрещает доступ к локальным файлам, таким как `../resources/sky.jpg`. В Firefox эквивалентное разрешение устанавливается с помощью параметра настройки `security.fileuri.strict_origin_policy` в `about:config`, которому должно быть присвоено значение `false`. Не забудьте вернуть его в прежнее значение после экспериментов, иначе вы оставите открытым доступ к локальным файлам, в обход системы безопасности.



Рис. 5.19. Приложение `TextureQuad` (слева) и использовавшееся изображение текстуры (справа)

В следующих разделах мы подробнее рассмотрим шаги с (1) по (4). Изображение, подготавливаемое на этапе (1), может иметь любой формат, поддерживаемый браузером. В своих экспериментах вы можете использовать любые изображения – и собственные, и хранящиеся на веб-сайте книги в папке `resource`.

Метод наложения текстуры, который требуется определить на этапе (2), описывает, «какая часть изображения текстуры» должна быть наложена и «на какую часть фигуры». Часть фигуры, предназначенная для покрытия текстурой, определяется путем определения координат вершин на поверхности фигуры. Часть изображения текстуры, которая будет использоваться, определяется **координатами текстуры**. Это новая разновидность координат, поэтому познакомимся с ними поближе.

Координаты текстуры

В WebGL используется 2-мерная система координат текстуры, как показано на рис. 5.20. Чтобы как-то отличать координаты текстуры от привычных координат X и Y , в WebGL оси координат текстуры были названы s и t (так называемая система координат st).³

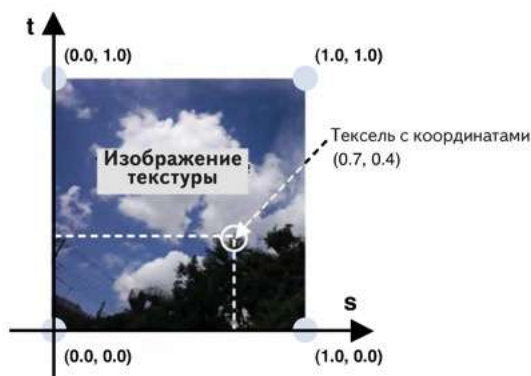


Рис. 5.20. Система координат текстуры в WebGL

Здесь точка текстуры с координатами $(0.0, 1.0)$ отображается в вершину с координатами $(-0.5, 0.5, 0.0)$, а точка текстуры с координатами $(1.0, 1.0)$ – в вершину с координатами $(0.5, 0.5, 0.0)$. Установив соответствие всех четырех углов изображения текстуры, вы получите результат, изображенный на рис. 5.21 справа.

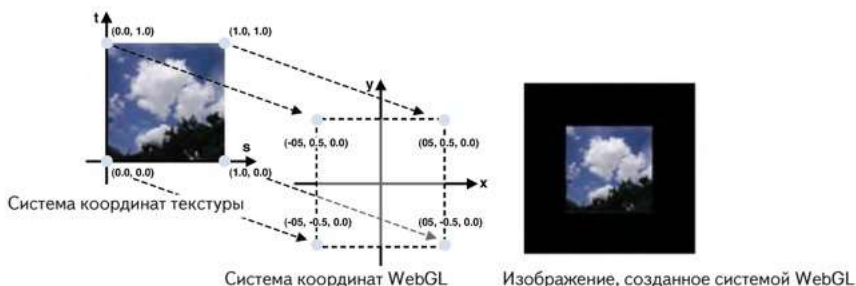


Рис. 5.21. Координаты текстуры и их наложение на вершины

Теперь, узнав, как накладываются изображения текстур на геометрические фигуры, рассмотрим пример программы.

Пример программы (TexturedQuad.js)

В программе `TexturedQuad.js` (см. листинг 5.7) реализация наложения текстуры затронула оба шейдера – вершинный и фрагментный. Это объясняется необходи-

³ Нередко ее называют также системой координат uv . Однако мы будем использовать название st , потому что именно эти имена осей (s и t) используются в языке GLSL ES.

мостью определения координат точек текстуры для каждой вершины и последующего применения соответствующих цветов пикселей, извлеченных из изображения текстуры для каждого фрагмента. Пример делится на пять основных частей, каждая из них подписана числовой меткой справа в листинге.

Листинг 5.7. TexturedQuad.js

```
1 // TexturedQuad.js
2 // Вершинный шейдер                                     <- (Часть 1)
3 var VSHADER_SOURCE =
4     'attribute vec4 a_Position;\n' +
5     'attribute vec2 a_TexCoord;\n' +
6     'varying vec2 v_TexCoord;\n' +
7     'void main() {\n' +
8     '    gl_Position = a_Position;\n' +
9     '    v_TexCoord = a_TexCoord;\n' +
10    '}\n';
11
12 // Фрагментный шейдер                                   <- (Часть 2)
13 var FSHADER_SOURCE =
14     ...
17     'uniform sampler2D u_Sampler;\n' +
18     'varying vec2 v_TexCoord;\n' +
19     'void main() {\n' +
20     '    gl_FragColor = texture2D(u_Sampler, v_TexCoord);\n' +
21     '}\n';
22
23 function main() {
24     ...
40     // Определить информацию о вершинах                 <- (Часть 3)
41     var n = initVertexBuffers(gl);
42     ...
50     // Определить текстуры
51     if (!initTextures(gl, n)) {
52         ...
54     }
55 }
56
57 function initVertexBuffers(gl) {
58     var verticesTexCoords = new Float32Array([
59         // Координаты вершин, координаты текстуры
60         -0.5, 0.5, 0.0, 1.0,
61         -0.5, -0.5, 0.0, 0.0,
62         0.5, 0.5, 1.0, 1.0,
63         0.5, -0.5, 1.0, 0.0,
64     ]);
65     var n = 4; // Число вершин
66
67     // Создать буферный объект
68     var vertexTexCoordBuffer = gl.createBuffer();
69     ...
74     // Записать координаты вершин и текстуры в буферный объект
75     gl.bindBuffer(gl.ARRAY_BUFFER, vertexTexCoordBuffer);
76     gl.bufferData(gl.ARRAY_BUFFER, verticesTexCoords, gl.STATIC_DRAW);
77 }
```

```

78   var FSIZE = verticesTexCoords.BYTES_PER_ELEMENT;
   ...
85   gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, FSIZE * 4, 0);
86   gl.enableVertexAttribArray(a_Position); // Разрешить использование буфера
87
88   // Записать координаты текстуры в переменную a_TexCoord и разрешить ее
89   var a_TexCoord = gl.getAttribLocation(gl.program, 'a_TexCoord');
   ...
94   gl.vertexAttribPointer(a_TexCoord, 2, gl.FLOAT, false, FSIZE*4, FSIZE*2);
95   gl.enableVertexAttribArray(a_TexCoord); // Разрешить использование буфера
   ...
97   return n;
98 }
99
100 function initTextures(gl, n)                                <- (Часть 4)
101   var texture = gl.createTexture(); // Создать объект текстуры
   ...
107   // Получить ссылку на u_Sampler
108   var u_Sampler = gl.getUniformLocation(gl.program, 'u_Sampler');
   ...
114   var image = new Image(); // Создать объект изображения
   ...
119   // Зарегистрировать обработчик, вызываемый после загрузки изображения
120   image.onload = function(){loadTexture(gl,n,texture,u_Sampler,image)};
121   // Заставить браузер загрузить изображение
122   image.src = '../resources/sky.jpg';
123
124   return true;
125 }
126
127 function loadTexture(gl, n, texture, u_Sampler, image){      <- (Часть 5)
128   gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, 1); // Повернуть ось Y изображения
129   // Выбрать текстурный слот 0
130   gl.activeTexture(gl.TEXTURE0);
131   // Указать тип объекта текстуры
132   gl.bindTexture(gl.TEXTURE_2D, texture);
133
134   // Определить параметры текстуры
135   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
136   // Определить изображение текстуры
137   gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGB, gl.RGB, gl.UNSIGNED_BYTE, image);
138
139   // Определить указатель на текстурный слот 0
140   gl.uniform1i(u_Sampler, 0);
   ...
144   gl.drawArrays(gl.TRIANGLE_STRIP, 0, n); // Нарисовать прямоугольник
145 }

```

Эта программа делится на пять основных частей:

- Часть 1:** прием координат текстуры в вершинный шейдер и передача их во фрагментный шейдер.
- Часть 2:** наложение изображения текстуры на геометрическую фигуру во фрагментном шейдере.
- Часть 3:** определение координат текстуры (`initVertexBuffers()`).

Часть 4: подготовка к загрузке изображения текстуры и передача браузеру требования загрузить его (`initTextures()`).

Часть 5: настройка загруженной текстуры к использованию в WebGL (`loadTexture()`).

Рассмотрим последовательность действий, начиная с части 3: процедуры определения координат текстуры с помощью `initVertexBuffers()`. Шейдеры выполняются после загрузки изображения, поэтому мы исследуем их в конце.

Использование координат текстуры (`initVertexBuffers()`)

Координаты текстуры можно передать в вершинный шейдер, используя тот же прием, что использовался для передачи других данных в вершинный шейдер, путем объединения координат вершин с другими данными в общем буфере. В строке 58 определяется массив `verticesTexCoords`, содержащий пары координат вершин и соответствующих им координат текстуры:

```
58 var verticesTexCoords = new Float32Array([
59     // Координаты вершин, координаты текстуры
60     -0.5, 0.5, 0.0, 1.0,
61     -0.5, -0.5, 0.0, 0.0,
62     0.5, 0.5, 1.0, 1.0,
63     0.5, -0.5, 1.0, 0.0,
64 ]);
```

Как видите, первой вершине $(-0.5, 0.5)$ соответствует точка текстуры с координатами $(0.0, 1.0)$, второй вершине $(-0.5, -0.5)$ – точка текстуры с координатами $(0.0, 0.0)$, третьей $(0.5, 0.5)$ – точка текстуры с координатами $(1.0, 1.0)$ и четвертой $(0.5, -0.5)$ – точка текстуры с координатами $(1.0, 0.0)$. Эти соответствия показаны на рис. 5.21.

В строках с 75 по 86 координаты вершин и текстуры записываются в буферный объект, который в свою очередь присваивается переменной `a_Position`, после чего разрешается его использование. Вслед за этим, в строках с 89 по 94 извлекается ссылка на переменную-атрибут `a_TexCoord` и этой переменной присваивается буферный объект, содержащий координаты текстуры. Наконец, в строке 95 разрешается использование этого буферного объекта:

```
88 // Записать координаты текстуры в переменную a_TexCoord и разрешить ее
89 var a_TexCoord = gl.getAttribLocation(gl.program, 'a_TexCoord');
...
94 gl.vertexAttribPointer(a_TexCoord, 2, gl.FLOAT, false, FSIZE*4, FSIZE*2);
95 gl.enableVertexAttribArray(a_TexCoord); // Разрешить использование буфера
```

Подготовка и загрузка изображений (`initTextures()`)

Эта процедура выполняется в строках со 101 по 122, в функции `initTextures()`. В строке 101 создается объект текстуры (`gl.createTexture()`), который будет

управлять изображением текстуры в системе WebGL, и в строке 108 извлекается ссылка на `uniform-переменную` (`gl.getUniformLocation()`) для передачи изображения текстуры во фрагментный шейдер:

```
101 var texture = gl.createTexture(); // Создать объект текстуры
...
108 var u_Sampler = gl.getUniformLocation(gl.program, 'u_Sampler');
```

Объект текстуры создается с помощью метода `gl.createTexture()`.

`gl.createTexture()`

Создает объект текстуры для хранения изображения.

Параметры	нет	
Возвращаемое значение	непустое значение	Вновь созданный объект текстуры.
	<code>null</code>	Ошибка создания объекта текстуры.
Ошибки	нет	

Вызов этого метода создает объект текстуры в системе WebGL, как показано на рис. 5.22. Константы от `gl.TEXTURE0` до `gl.TEXTURE7` – это текстурные слоты, предназначенные для управления изображениями текстур, каждый из которых имеет тип текстуры `gl.TEXTURE_2D`. Но подробнее об этом мы поговорим позже.

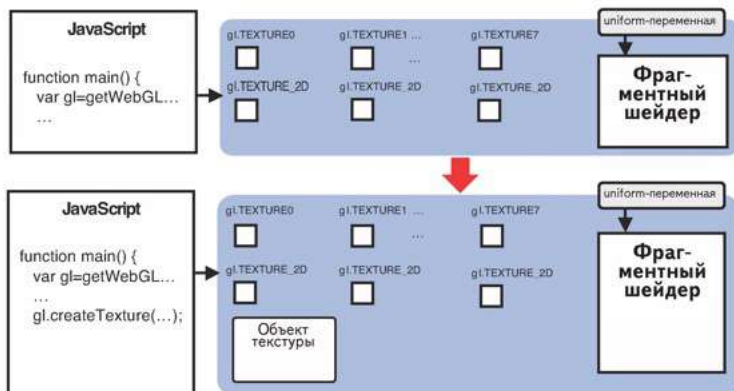


Рис. 5.22. Создание объекта текстуры

Объект текстуры можно удалить вызовом метода `gl.deleteTexture()`. Обратите внимание, что если этому методу передать ссылку на уже удаленный объект текстуры, он просто не выполнит никаких действий.

`gl.deleteTexture(texture)`

Удаляет объект текстуры `texture`.

Параметры	<code>texture</code>	Ссылка на объект текстуры, подлежащий удалению.
------------------	----------------------	---

Возвращаемое значение	нет
Ошибки	нет

На следующем шаге необходимо потребовать от браузера загрузить изображение, которое будет наложено на прямоугольник. Для этого необходимо создать объект `Image`:

```
114  var image = new Image(); // Создать объект изображения
    ...
119  // Зарегистрировать обработчик, вызываемый после загрузки изображения
120  image.onload = function(){loadTexture(gl,n,texture,u_Sampler,image)};
121  // Заставить браузер загрузить изображение
122  image.src = '../resources/sky.jpg';
```

Этот фрагмент кода создает объект `Image`, регистрирует обработчик события (`loadTexture()`) для вызова по окончании загрузки изображения и сообщает браузеру, что он должен загрузить изображение.

Объект `Image` (специальный объект JavaScript, используемый для работы с изображениями) создается с помощью оператора `new`, так же как объект `Array` или `Date`. Эта операция выполняется в строке 114.

```
114  var image = new Image(); // Создать объект изображения
```

Так как загрузка изображений выполняется асинхронно (см. врезку ниже), необходимо зарегистрировать обработчик, который автоматически будет вызван браузером по окончании загрузки. Этот обработчик должен передать изображение в систему WebGL. Регистрация обработчика выполняется в строке 120. Там же находится его реализация: анонимная функция вызывает `loadTexture()` и передает ей загруженное изображение.

```
120  image.onload = function(){loadTexture(gl,n,texture,u_Sampler,image)};
```

Функция `loadTexture()` принимает пять параметров. В последнем аргументе ей передается только что загруженное изображение (объект `Image`). В аргументе `gl` передается контекст отображения WebGL, в аргументе `n` – число вершин, в аргументе `texture` – объект текстуры, созданный в строке 101, и в аргументе `u_Sampler` – ссылка на `uniform`-переменную.

По аналогии с тегом `<img>` в языке разметки HTML, мы можем потребовать от браузера загрузить изображение текстуры простым присваиванием имени файла свойству `src` объекта `Image` (строка 122). Обратите внимание, что из-за обычных ограничений, связанных с безопасностью, браузер не может загружать изображения, находящиеся в других доменах:

```
122  image.src = '../resources/sky.jpg';
```

После выполнения строки 122 браузер выполняет загрузку изображения асинхронно, поэтому далее программа выполняет инструкцию `return` в строке

124 и завершается. Когда браузер загрузит изображение, он вызовет обработчик `loadTexture()`, который в свою очередь передаст изображение системе WebGL.

Асинхронная загрузка изображений текстур

Обычно приложения OpenGL, написанные на С или С++, загружают файлы изображений текстур непосредственно с жесткого диска. Однако, из-за того, что WebGL-программы выполняются под управлением браузера, они не могут загружать файлы напрямую. Вместо этого они вынуждены обращаться к браузеру, чтобы тот выполнил необходимую операцию. (Как правило, в таких случаях браузер посылает запрос веб-серверу и получает изображение от него.) Такой подход позволяет использовать любые изображения, поддерживаемые браузером, но он же делает весь процесс сложнее, потому что вынуждает разделить его на два процесса (передачу браузеру запроса на загрузку изображения и передачу загруженного изображения в систему WebGL), действующие «асинхронно» (они выполняются в фоновом режиме и не блокируют выполнение программы).

На рис. 5.23 показаны этапы между [1] отправкой браузеру запроса на загрузку изображения и [7] вызовом функции `loadTexture()`.

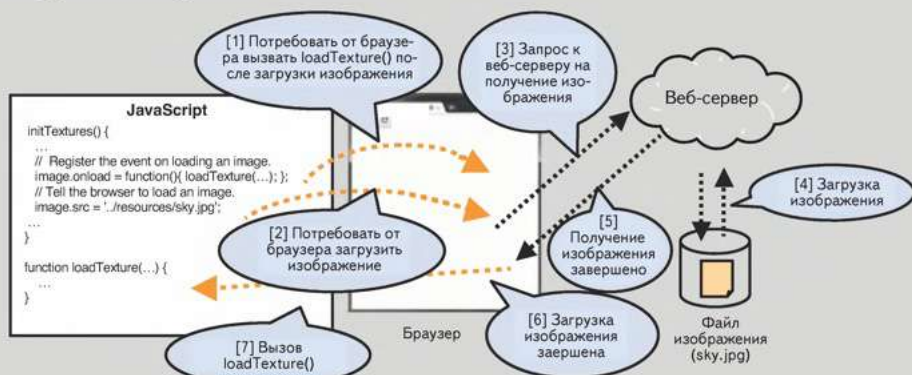


Рис. 5.23. Асинхронная загрузка изображения текстуры

Как показано на рис. 5.23, этапы [1] и [2] выполняются последовательно и синхронно, но этапы [2] и [7] – нет. После отправки браузеру запроса на загрузку изображения [2], программа на JavaScript не ждет, пока изображение будет загружено, а продолжает выполнение. (Такое ее поведение объясняется чуть ниже.) Пока программа на JavaScript продолжает выполнение, браузер посылает запрос веб-серверу на получение требуемого изображения [3]. Когда процесс получения [4] и загрузки [5] завершится, браузер сообщит программе на JavaScript, что изображение загружено [6]. Такое поведение называют **асинхронным**.

Процесс загрузки изображения из программы на JavaScript аналогичен процессу загрузки изображения в веб-странице, написанной на HTML. Чтобы в веб-странице отобразить изображение нужно указать URL изображения в атрибуте `src` тега `<img>` (см. ниже), что заставит браузер загрузить указанное изображение. Эта часть соответствует этапу [2] на рис. 5.23.

```
<img src='../resources/redflower.jpg'>
```

Асинхронную природу процесса загрузки изображений проще понять, посмотрев, как отображается веб-страница, включающая множество изображений. Обычно текст и разметка отображаются быстро, а изображения появляются с некоторой задержкой. Это объясняется тем, что процесс загрузки и отображения изображений выполняется асинхронно, позволяя вам немедленно приступить к взаимодействию со страницей, не дожидаясь, пока все изображения будут загружены.

Подготовка загруженной текстуры к использованию в WebGL (`loadTexture()`)

Ниже приводится определение функции `loadTexture()`:

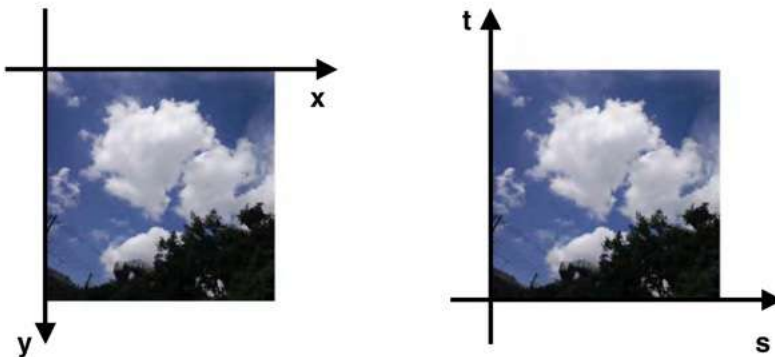
```
127 function loadTexture(gl, n, texture, u_Sampler, image){          <- (Часть 5)
128   gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, 1); // Повернуть ось Y изображения
129   // Выбрать текстурный слот 0
130   gl.activeTexture(gl.TEXTURE0);
131   // Указать тип объекта текстуры
132   gl.bindTexture(gl.TEXTURE_2D, texture);
133
134   // Определить параметры текстуры
135   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
136   // Определить изображение текстуры
137   gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGB, gl.RGB, gl.UNSIGNED_BYTE, image);
138
139   // Определить указатель на текстурный слот 0
140   gl.uniform1i(u_Sampler, 0);
141   ...
144   gl.drawArrays(gl.TRIANGLE_STRIP, 0, n); // Нарисовать прямоугольник
145 }
```

Ее главная цель состоит в том, чтобы подготовить изображение к использованию системой WebGL, которая достигается за счет настройки **объекта текстуры** и использования его в качестве буферного объекта. В следующих разделах реализация функции описывается более подробно.

Поворот оси Y изображения

Прежде чем загруженное изображение можно будет использовать в качестве текстуры, необходимо повернуть ось Y в противоположном направлении:

```
128   gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, 1); // Повернуть ось Y изображения
```



Система координат изображения

Система координат WebGL

Рис. 5.24. Системы координат изображения и текстуры в WebGL

Этот метод поворачивает ось Y изображения. Как показано на рис. 5.24, ось T в системе координат текстуры направлена в противоположную сторону, в сравнении с осью Y в системе координат, используемой в форматах PNG, BMP, JPG и других. По этой причине корректное наложение изображения на фигуру возможно только после поворота его оси Y в обратном направлении. (Можно также вручную преобразовывать координаты t.)

Ниже приводится описание метода `gl.pixelStorei()`.

`gl.pixelStorei(pname, param)`

Выполняет операцию, определяемую параметрами `pname` и `param`, после загрузки изображения.

Параметры	<code>pname</code>	Может принимать одно из следующих значений:
	<code>gl.UNPACK_FLIP_Y_WEBGL</code>	Поворачивает ось Y изображения. Значение по умолчанию <code>false</code> .
	<code>gl.UNPACK_PREMULTIPLY_ALPHA_WEBGL</code>	Умножает каждую составляющую цвета в формате RGB на значение составляющей A (альфа-канала). Значение по умолчанию <code>false</code> .
	<code>param</code>	Целочисленное ненулевое значение соответствует <code>true</code> , нулевое значение соответствует <code>false</code> .
Возвращаемое значение	нет	
Ошибки	<code>INVALID_ENUM</code>	В параметре <code>pname</code> передано недопустимое значение, не являющееся ни одним из перечисленных выше.

Выбор текстурного слота (`gl.activeTexture()`)

Система WebGL поддерживает возможность использования сразу нескольких текстур, предоставляя механизм, который называется *текстурные слоты*. С помощью текстурных слотов можно управлять текстурами, используя порядковые номера слотов. Вследствие этого, даже если вам нужна всего одна текстура, вы должны указывать и использовать текстурный слот.

Число поддерживаемых текстурных слотов зависит от аппаратного обеспечения и реализации WebGL, но не менее восьми, а в некоторых системах их значительно больше. Слоты представлены встроенными константами `gl.TEXTURE0`, `gl.TEXTURE1`, ..., `gl.TEXTURE7` (см. рис. 5.25).

Прежде чем использовать текстурный слот, его необходимо выбрать (активизировать) вызовом `gl.activeTexture()` (см. рис. 5.26):

```
129 // Выбрать текстурный слот 0
130 gl.activeTexture(gl.TEXTURE0);
```

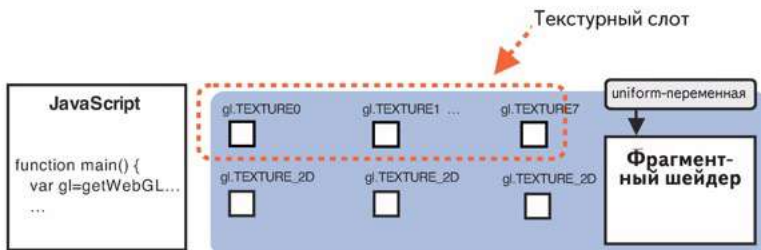


Рис. 5.25. Множество текстурных слотов, поддерживаемых системой WebGL

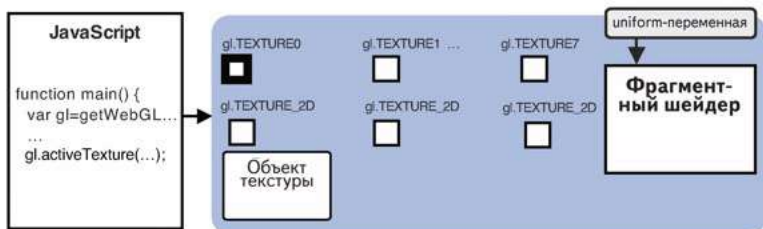


Рис. 5.26. Выбор текстурного слота (gl.TEXTURE0)

gl.activeTexture (texUnit)Выбирает (активизирует) текстурный слот `texUnit`.

Параметры `texUnit` Определяет выбираемый текстурный слот: `gl.TEXTURE0`, `gl.TEXTURE1`, ..., `gl.TEXTURE7`. Число в конце имени соответствует порядковому номеру текстурного слота.

Возвращаемое значение нет

Ошибки `INVALID_ENUM` В параметре `texUnit` передано недопустимое значение, не являющееся ни одним из перечисленных выше.

Указание типа объекта текстуры (*gl.bindTexture()*)

Далее необходимо сообщить системе WebGL тип изображения текстуры в объекте текстуры. Делается это путем указания типа объекта текстуры, подобно тому, как это делалось для буферного объекта в предыдущей главе. WebGL поддерживает два типа текстур, перечисленных в табл. 5.2.

Таблица 5.2. Типы текстур

Тип текстуры	Описание
<code>gl.TEXTURE_2D</code>	2-мерная (плоская) текстура
<code>gl.TEXTURE_CUBE_MAP</code>	Кубическая текстура

В примере программы в качестве текстуры используется 2-мерное изображение, поэтому в строке 132 мы указываем тип `gl.TEXTURE_2D`. Обсуждение кубических текстур далеко выходит за рамки этой книги, поэтому мы не будем их рассматривать.⁴ Желаящим получить дополнительную информацию мы можем порекомендовать книгу «OpenGL ES 2.0 Programming Guide»:

```
131 // Указать тип объекта текстуры
132 gl.bindTexture(gl.TEXTURE_2D, texture);
```

`gl.bindTexture(target, texture)`

Активизирует объект текстуры `texture` и указывает его тип `target`. Кроме того, если прежде, вызовом `gl.activeTexture()`, был активизирован текстурный слот, объект текстуры также будет связан с активным слотом.

Параметры	<code>target</code>	Может иметь значение <code>gl.TEXTURE_2D</code> или <code>gl.TEXTURE_CUBE_MAP</code> .
	<code>texture</code>	Объект текстуры, тип которого требуется указать.
Возвращаемое значение	нет	
Ошибки	<code>INVALID_ENUM</code>	В параметре <code>target</code> передано недопустимое значение, не являющееся ни одним из перечисленных выше.

Обратите внимание, что этот метод решает две задачи: активизирует объект текстуры и определяет его тип, а также связывает его с активным текстурным слотом. В данном случае, так как активным является слот с номером 0 (`gl.TEXTURE0`), после выполнения строки 132 внутреннее состояние системы WebGL изменится, как показано на рис. 5.27.

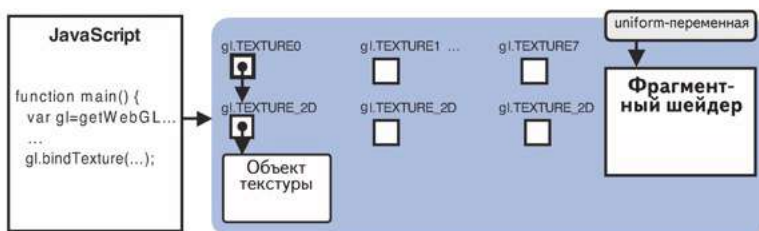


Рис. 5.27. Указание типа объекта текстуры

На данный момент программа определила тип текстуры в объекте текстуры (`gl.TEXTURE_2D`), который будет использоваться для работы с текстурой в будущем. Это важно, потому что в WebGL невозможно манипулировать объектом текстуры непосредственно. Это возможно только через определение типа.

⁴ Кубическая текстура по сути является набором из шести текстур одинакового размера, предназначенных для наложения на грани куба. – Прим. перев.

Настройка параметров объекта текстуры (*gl.texParameteri()*)

Далее нужно задать параметры (параметры текстуры), определяющие порядок обработки изображения текстуры при наложении ее на фигуры. Для настройки параметров можно использовать метод `gl.texParameteri()`.

`gl.texParameteri(target, pname, param)`

Присваивает значение `param` параметру текстуры `pname` в объекте текстуры с типом `target`.

Параметры	<code>target</code>	Может иметь значение <code>gl.TEXTURE_2D</code> или <code>gl.TEXTURE_CUBE_MAP</code> .
	<code>pname</code>	Имя параметра текстуры (см. табл. 5.3).
	<code>param</code>	Значение параметра с именем <code>pname</code> (см. табл. 5.4 и табл. 5.5).
Возвращаемое значение	нет	
Ошибки	<code>INVALID_ENUM</code>	В параметре <code>target</code> передано недопустимое значение, не являющееся ни одним из перечисленных выше.
	<code>INVALID_OPERATION</code>	Нет объекта текстуры с типом <code>target</code> .

Текстуры имеют четыре параметра, как показано на рис. 5.28, имена которых можно передать в аргументе `pname`:

- **метод увеличения** (`gl.TEXTURE_MAG_FILTER`): метод увеличения изображения текстуры, когда размеры фигуры, на которую она накладывается, оказываются больше размеров текстуры. Например, когда изображение размером 16×16 пикселей накладывается на фигуру размером 32×32 пикселя, размеры изображения текстуры требуется увеличить вдвое. Системе WebGL необходимо как-то заполнить пропуски между текселями, появление которых обусловлено увеличением. Этот параметр как раз определяет метод заполнения;
- **метод уменьшения** (`gl.TEXTURE_MIN_FILTER`): метод уменьшения изображения текстуры, когда размеры фигуры, на которую она накладывается, оказываются меньше размеров текстуры. Например, когда изображение размером 32×32 пикселей накладывается на фигуру размером 16×16 , размеры изображения текстуры требуется уменьшить. Для этого системе WebGL необходимо удалить часть текселей, чтобы привести изображение к требуемым размерам. Этот параметр как раз определяет метод выбора удаляемых текселей.
- **метод заполнения слева и справа** (`gl.TEXTURE_WRAP_S`): этот параметр определяет, как должны заполняться пустые области слева и справа от подобласти фигуры, на которую наложена текстура.

- **метод заполнения сверху и снизу** (`gl.TEXTURE_WRAP_T`): этот параметр напоминает предыдущий и определяет, как должны заполняться пустые области сверху и снизу от подобласти фигуры, на которую наложена текстура.

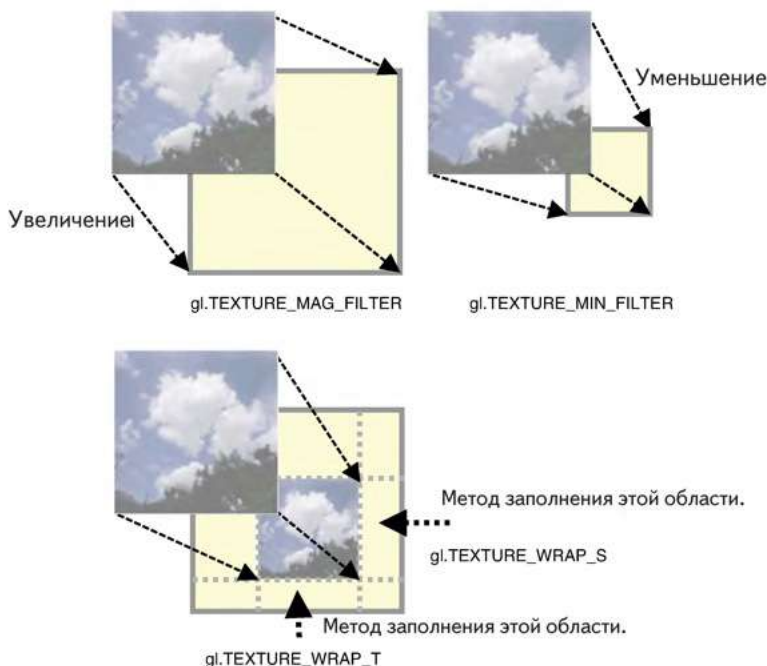


Рис. 5.28. Четыре параметра текстур и их действие

В табл. 5.3 перечислены все параметры текстуры и их значения по умолчанию.

Таблица 5.3. Параметры текстуры и их значения по умолчанию

Параметр	Описание	Значение по умолчанию
<code>gl.TEXTURE_MAG_FILTER</code>	Определяет метод увеличения текстуры.	<code>gl.LINEAR</code>
<code>gl.TEXTURE_MIN_FILTER</code>	Определяет метод уменьшения текстуры.	<code>gl.NEAREST_MIPMAP_LINEAR</code>
<code>gl.TEXTURE_WRAP_S</code>	Определяет метод заполнения областей по оси S.	<code>gl.REPEAT</code>
<code>gl.TEXTURE_WRAP_T</code>	Определяет метод заполнения областей по оси T.	<code>gl.REPEAT</code>

Кроме того, в табл. 5.4 описываются константы, которые можно использовать в качестве значений параметров `gl.TEXTURE_MAG_FILTER` и `gl.TEXTURE_MIN_FILTER`, а в табл. 5.5 – константы, которые можно использовать в качестве значений параметров `gl.TEXTURE_WRAP_S` и `gl.TEXTURE_WRAP_T`.

Таблица 5.4. Некоторые допустимые значения параметров `gl.TEXTURE_MAG_FILTER` и `gl.TEXTURE_MIN_FILTER`⁵

Значение	Описание
<code>gl.NEAREST</code>	Используется значение текселя, ближайшего (ближайшего, в смысле алгоритма «городских кварталов». (Manhattan distance)) к центру текстурируемого пикселя.
<code>gl.LINEAR</code>	Используется среднее взвешенное по четырем текселям, ближайшим к центру текстурируемого пикселя. (Этот метод обеспечивает более высокое качество, но требует большего объема вычислений и, соответственно, времени.)

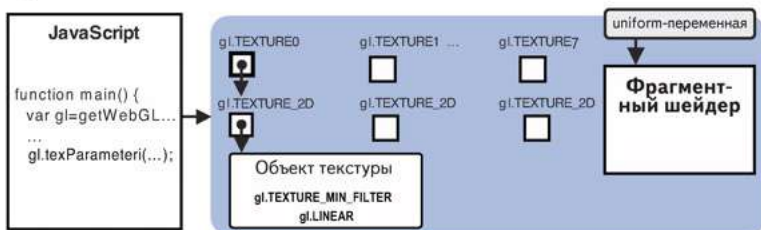
Таблица 5.5. Допустимые значения параметров `gl.TEXTURE_WRAP_S` и `gl.TEXTURE_WRAP_T`

Значение	Описание
<code>gl.REPEAT</code>	Использовать изображение текстуры повторно.
<code>gl.MIRRORED_REPEAT</code>	Использовать изображение текстуры повторно с отражением.
<code>gl.CLAMP_TO_EDGE</code>	Использовать цвет края изображения текстуры.

Как показано в табл. 5.3, Каждый параметр имеет значение по умолчанию, которое с успехом можно использовать в большинстве ситуаций. Однако, значение по умолчанию для параметра `gl.TEXTURE_MIN_FILTER` стоит особняком и предназначено для случаев использования текстур в формате, который называется *MIPMAP*. Текстура в формате MIPMAP (или MIP-текстура) представляет собой последовательность текстур, расположенных в порядке уменьшения разрешения одного и того же изображения. Так как MIP-текстуры используются не особенно часто, мы не будем рассматривать их в этой книге. По этой причине в строке 135 мы присвоили параметру `gl.TEXTURE_MIN_FILTER` значение `gl.LINEAR`:

```
134 // Определить параметры текстуры
135 gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
```

После выполнения строки 139 параметру объекта текстуры будет присвоено указанное значение, и внутреннее состояние системы WebGL изменится, как показано на рис. 5.29.

**Рис. 5.29.** Установка значения параметра объекта текстуры

⁵ В этой таблице опущены значения, имеющие отношение к MIP-текстурированию (MIPMAP texture): `gl.NEAREST_MIPMAP_NEAREST`, `gl.LINEAR_MIPMAP_NEAREST`, `gl.NEAREST_MIPMAP_LINEAR` и `gl.LINEAR_MIPMAP_LINEAR`. Описание этих значений можно найти в книге «OpenGL Programming Guide».

Следующий шаг – присваивание изображения текстуры объекту текстуры.

Присваивание изображения объекту текстуры (*gl.texImage2D()*)

Присваивание изображения объекту текстуры выполняется с помощью метода `gl.texImage2D()`. Кроме присваивания изображения, этот метод позволяет также сообщить системе WebGL дополнительные характеристики изображения.

`gl.texImage2D(target, level, internalformat, format, type, image)`

Присваивает изображение `image` объекту текстуры с типом `target`.

Параметры	<code>target</code>	Может иметь значение <code>gl.TEXTURE_2D</code> или <code>gl.TEXTURE_CUBE_MAP</code> .
	<code>level</code>	В этом параметре передается значение 0. (В действительности этот параметр используется совместно с MIP-текстурами, которые не рассматриваются в данной книге.)
	<code>internalformat</code>	Определяет внутренний формат изображения (см. табл. 5.6).
	<code>format</code>	Определяет формат данных с информацией о текселях. Должен иметь то же значение, что и <code>internalformat</code> .
	<code>type</code>	Определяет тип данных с информацией о текселях (см. табл. 5.7).
	<code>image</code>	Объект изображения, содержащий текстуру.

Возвращаемое значение нет

Ошибки	<code>INVALID_ENUM</code>	В параметре <code>target</code> передано недопустимое значение, не являющееся ни одним из перечисленных выше.
	<code>INVALID_OPERATION</code>	Нет объекта текстуры с типом <code>target</code> .

Этот метод вызывается в строке 137:

```
137 gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGB, gl.RGB, gl.UNSIGNED_BYTE, image);
```

После выполнения строки 137 изображение текстуры, загруженное в объект `Image`, будет передано в систему WebGL (см. рис. 5.30).

Рассмотрим вкратце каждый параметр этого метода. В параметре `level` необходимо передавать значение 0, потому что мы не используем MIP-текстуры. Параметр `format` определяет формат данных с информацией о текселях. Поддерживаемые форматы перечислены в табл. 5.6. Вы должны выбрать формат, соответствующий используемому изображению. В примере программы указан формат `gl.RGB`, потому что он используется в изображениях JPG, где каждый пиксель представлен компонентами RGB. В некоторых других форматах, таких как PNG, обычно используется формат `gl.RGBA`. Для изображений BMP использует-

ся формат `gl.RGB`. Для черно-белых изображений часто используются форматы `gl.LUMINANCE` и `gl.LUMINANCE_ALPHA`, и так далее.

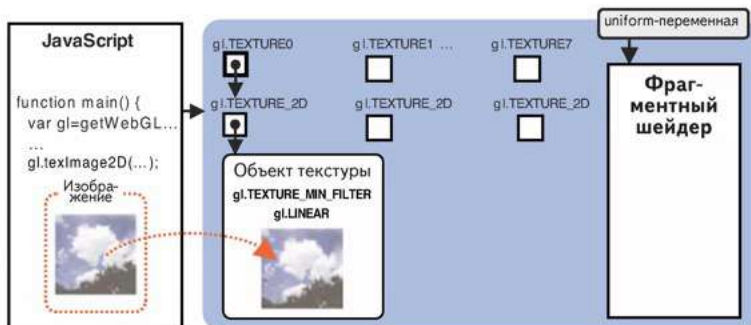


Рис. 5.30. Присваивание изображения объекту текстуры

Таблица 5.6. Форматы данных с информацией о текстурах

Формат	Составляющие цвета в текстурах
<code>gl.RGB</code>	Красный, зеленый, синий (red, green, blue).
<code>gl.RGBA</code>	Красный, зеленый, синий, альфа-канал (red, green, blue, alpha).
<code>gl.ALPHA</code>	(0.0, 0.0, 0.0, alpha)
<code>gl.LUMINANCE</code>	(L, L, L, 1), где L – Luminance (светимость)
<code>gl.LUMINANCE_ALPHA</code>	(L, L, L, alpha)

Здесь под «светимостью» понимается воспринимаемая яркость поверхности. Часто вычисляется как взвешенное среднее значений красной, зеленой и синей составляющих.

Как показано на рис. 5.30, этот метод сохраняет изображение в объекте текстуры, внутри системы WebGL. При сохранении необходимо также сообщить системе тип формата используемого изображения в параметре `internalformat`. Как уже упоминалось, в WebGL параметр `internalformat` должен иметь то же значение, что и параметр `format`.

Параметр `type` определяет тип данных с информацией о текстурах (см. табл. 5.7). Обычно в этом параметре передается значение `gl.UNSIGNED_BYTE`. Поддерживаются также другие типы данных, такие как `gl.UNSIGNED_SHORT_5_6_5` (в котором компоненты RGB упакованы в 16-битное число). Эти типы используются для передачи в WebGL сжатых изображений, чтобы уменьшить время загрузки.

Таблица 5.7. Типы данных с информацией о текстурах

Тип	Описание
<code>gl.UNSIGNED_BYTE</code>	Компоненты RGB представлены беззнаковыми байтами – каждый компонент представлен одним байтом.
<code>gl.UNSIGNED_SHORT_5_6_5</code>	RGB: компоненты RGB представлены 5, 6 и 5 битами, соответственно.

Тип	Описание
<code>gl.UNSIGNED_SHORT_4_4_4_4</code>	RGBA: компоненты RGBA представлены 4, 4, 4 и 4 битами, соответственно.
<code>gl.UNSIGNED_SHORT_5_5_5_1</code>	RGBA: компоненты RGB представлены 5 битами, а альфа-канал – одним битом.

Передача текстурного слота фрагментному шейдеру (`gl.uniform1i()`)

После передачи изображения текстуры в систему WebGL, его необходимо передать фрагментному шейдеру, для наложения на поверхность фигуры. Как описывалось выше, для этой цели используется `uniform`-переменная, потому что для всех фрагментов изображение остается одним и тем же:

```
13 var FSHADER_SOURCE =
    ...
17 'uniform sampler2D u_Sampler;\n' +
18 'varying vec2 v_TexCoord;\n' +
19 'void main() {\n' +
20 '    gl_FragColor = texture2D(u_Sampler, v_TexCoord);\n' +
21 '}\n';
```

Эта `uniform`-переменная должна быть объявлена с использованием одного из специальных типов данных, предназначенных для текстур (см. табл. 5.8). В примере программы используется 2-мерная текстура (`gl.TEXTURE_2D`), поэтому мы использовали тип `sampler2D`.

Таблица 5.8. Специальные типы данных для текстур

Тип	Описание
<code>sampler2D</code>	Тип данных для текстур типа <code>gl.TEXTURE_2D</code> .
<code>samplerCube</code>	Тип данных для текстур типа <code>gl.TEXTURE_CUBE_MAP</code> .

Функция `initTextures()` (начало определения находится в строке 100) получает ссылку на `uniform`-переменную `u_Sampler` (строка 108) и затем передает ее в вызов `loadTexture()`. В строке 140 `uniform`-переменной `u_Sampler` присваивается номер текстурного слота («n» в имени `gl.TEXTUREn`), управляющего данным объектом текстуры. В нашем примере программы мы используем число 0, потому что связали объект текстуры со слотом `gl.TEXTURE0` в вызове `gl.uniform1i()`:

```
139 // Определить указатель на текстурный слот 0
140 gl.uniform1i(u_Sampler, 0);
```

После выполнения строки 139 внутреннее состояние системы WebGL изменится, как показано на рис. 5.31, что обеспечит доступ к изображению в объекте текстуры внутри фрагментного шейдера.

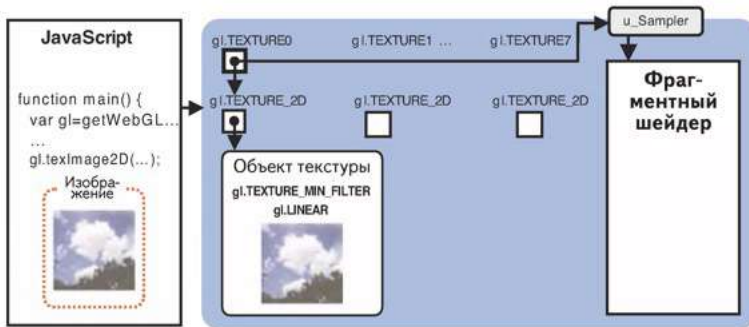


Рис. 5.31. Передача текстурного слота через uniform-переменную

Передача координат текстуры из вершинного шейдера во фрагментный шейдер

Так как координаты текстуры, соответствующие каждой вершине, передаются через переменную-атрибут `a_TexCoord`, есть возможность передать данные во фрагментный шейдер через `varying`-переменную `v_TexCoord`. Напомним, что значения `varying`-переменных с теми же именами и типами автоматически копируются между вершинным и фрагментным шейдерами. Координаты текстуры интерполируются между вершинами, благодаря чему во фрагментном шейдере мы можем использовать уже интерполированные координаты:

```

2 // Вершинный шейдер                                     <- (Часть 1)
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'attribute vec2 a_TexCoord;\n' +
6   'varying vec2 v_TexCoord;\n' +
7   'void main() {\n' +
8   '   gl_Position = a_Position;\n' +
9   '   v_TexCoord = a_TexCoord;\n' +
10  '}\n';

```

На этом подготовку изображения текстуры к использованию в системе WebGL можно считать законченной.

Теперь нам осталось лишь прочесть цвет текселя текстуры с указанными координатами и использовать его для установки цвета фрагмента.

Извлечение цвета текселя во фрагментном шейдере (`texture2D()`)

Извлечение цвета текселя из изображения выполняется в строке 20, внутри фрагментного шейдера:

```

20   '   gl_FragColor = texture2D(u_Sampler, v_TexCoord);\n' +

```


Здесь используется встроенная в язык GLSL ES функция `texture2D()`, которая возвращает цвет текселя с указанными координатами. Функция `texture2D()` достаточно проста в использовании и возвращает цвет текселя в текстуре, определяемой номером слота в первом аргументе, и с координатами, указанными во втором аргументе. Однако, так как эта функция является встроенной функцией языка GLSL ES, обратите внимание на типы ее параметров и возвращаемого значения.

vec4 texture2D(sampler2D sampler, vec2 coord)		
Возвращает цвет текселя в текстуре <code>sampler</code> с координатами <code>coord</code> .		
Параметры	<code>sampler</code>	Определяет номер текстурного слота.
	<code>coord</code>	Определяет координаты текселя текстуры.
Возвращаемое значение	Цвет текселя (<code>vec4</code>) с указанными координатами. Формат представления цвета зависит от значения параметра <code>internalformat</code> , указанного в вызове метода <code>gl.texImage2D()</code> . Различия показаны в табл. 5.9. Если по каким-то причинам изображение текстуры недоступно, эта функция вернет (0.0, 0.0, 0.0, 1.0).	

Таблица 5.9. Возвращаемые значения функции `texture2D()`

Значение параметра <code>internalformat</code>	Возвращаемое значение
<code>gl.RGB</code>	(R, G, B, 1.0)
<code>gl.RGBA</code>	(R, G, B, A)
<code>gl.ALPHA</code>	(0.0, 0.0, 0.0, A)
<code>gl.LUMINANCE</code>	(L, L, L, 1.0), где L – Luminance (светимость)
<code>gl.LUMINANCE_ALPHA</code>	(L, L, L, A)

Параметры увеличения и уменьшения текстуры определяют возвращаемое значение в случаях, когда WebGL выполняет интерполяцию текселя. После выполнения этой функции, ее возвращаемое значение можно присвоить переменной `gl_FragColor`, чтобы отобразить фрагмент с данным цветом. В результате этой операции изображение текстуры накладывается на фигуру (в данном случае на прямоугольник).

Это был последний этап, который необходимо выполнить для наложения текстуры. На этой стадии изображение текстуры загружено, передано в систему WebGL и наложено на фигуру.

Как было показано выше, наложение текстуры в WebGL является довольно сложной процедурой, отчасти из-за необходимости требовать от браузера загрузить его, а отчасти из-за того, что приходится использовать текстурный слот, даже при использовании единственной текстуры. Однако, овладев всеми этапами, вы без особого труда будете воспроизводить их снова и снова, так как сама последовательность действий, связанных с наложением текстуры, не изменяется.

В следующем разделе мы исследуем использование текстур и познакомимся со всем процессом в целом.

Эксперименты с примером программы

Чтобы лучше понять, как выполняется наложение текстуры, поэкспериментируем с примером программы и попробуем изменить координаты текстуры, как показано ниже:

```
var verticesTexCoords = new Float32Array([
    // Координаты вершин и текстуры
    -0.5,  0.5, -0.3,  1.7,
    -0.5, -0.5, -0.3, -0.2,
    0.5,  0.5,  1.7,  1.7,
    0.5, -0.5,  1.7, -0.2
]);
```

Если загрузить в браузер измененную версию `TexturedQuad_Repeat` программы, можно увидеть результат, как показано на рис. 5.32 (слева). Чтобы понять, как получился такой результат, взгляните на рис. 5.32 справа, где показаны все координаты текстуры в ее системе координат.

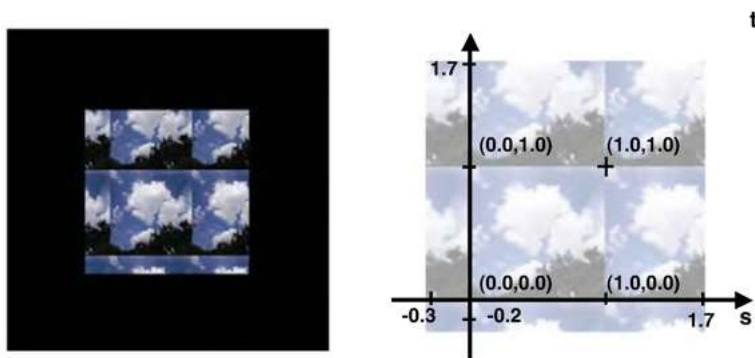


Рис. 5.32. Результат изменения координат текстуры (скриншот программы `TexturedQuad_Repeat`)

Изображение имеет недостаточные размеры, чтобы покрыть всю фигуру, поэтому, как видно на рис. 5.32, изображение накладывается несколько раз. Такое наложение обеспечивается присваиванием значения `gl.REPEAT` параметрами `gl.TEXTURE_WRAP_S` и `gl.TEXTURE_WRAP_T`, которое требует от системы WebGL повторять наложение текстуры до покрытия всей площади фигуры.

Теперь изменим параметры текстуры, как показано ниже, чтобы увидеть, какие еще эффекты нам доступны. Измененную программу мы сохранили под именем `TexturedQuad_Clamp_Mirror`, а на рис. 5.33 показан результат ее работы:

```
// Установить параметры текстуры
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR );
```

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE );  
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.MIRRORED_REPEAT );
```



Рис. 5.33. TexturedQuad_Clamp_Mirror

На рисунке видно, что по оси S (по горизонтали) повторяется вывод текселей, цвет которых совпадает с цветом края изображения, а по оси T (по вертикали) выводятся повторяющиеся изображения в зеркальном отражении.

На этом мы завершаем исследование простых приемов наложения текстур, доступных в WebGL. В следующем разделе мы посмотрим, как, опираясь на эти простые приемы, реализовать наложение сразу нескольких текстур.

Наложение нескольких текстур на фигуру

Выше в этой главе мы узнали, что WebGL может использовать сразу несколько изображений текстур, что объясняет поддержку нескольких текстурных слотов. В примерах, приводившихся до сих пор, мы использовали лишь одно изображение текстуры и, соответственно, один текстурный слот. В этом разделе мы рассмотрим пример программы `MultiTexture`, накладывающей на прямоугольник два изображения текстур, что позволит нам лучше понять работу механизма текстурных слотов. На рис. 5.34 показан результат выполнения программы `MultiTexture`. Как



Рис. 5.34. MultiTexture

можно видеть на этом скриншоте, программа «смешивает» два изображения текстур для получения сложного визуального эффекта.

На рис. 5.35 показаны две текстуры по отдельности, используемые в этом примере программы. Чтобы подчеркнуть возможность WebGL работать с изображениями в разных форматах, в этом примере преднамеренно были использованы разные типы изображений.



Рис. 5.35. Изображения текстур (слева: sky.jpg; справа: circle.gif), используемые в программе MultiTexture

Наложить несколько текстур на фигуру можно, просто повторяя процесс наложения единственной текстуры, описанный в предыдущем разделе. Давайте исследуем пример программы и посмотрим, как это делается.

Пример программы (MultiTexture.js)

В листинге 5.8 приводится содержимое файла MultiTexture.js, напоминающее содержимое файла TexturedQuad.js за исключением трех важных отличий: (1) фрагментный шейдер работает сразу с двумя текстурами, (2) цвет фрагмента вычисляется на основе двух текстелей из двух текстур, и (3) функция initTextures() создает два объекта текстуры.

Листинг 5.8. MultiTexture.js

```
1 // MultiTexture.js
...
13 var FSHADER_SOURCE =
...
17 'uniform sampler2D u_Sampler0;\n' +
18 'uniform sampler2D u_Sampler1;\n' +
19 'varying vec2 v_TexCoord;\n' +
20 'void main() {\n' +
21 '    vec4 color0 = texture2D(u_Sampler0, v_TexCoord);\n' +
22 '    vec4 color1 = texture2D(u_Sampler1, v_TexCoord);\n' +
23 '    gl_FragColor = color0 * color1;\n' +
24 '}\n';
25
26 function main() {
...
    <--(1)
    <--(2)
```

```

53 // Определить текстуры
54 if (!initTextures(gl, n)) {
55     ...
56 }
57
60 function initVertexBuffers(gl) {
61     var verticesTexCoords = new Float32Array([
62         // Координаты вершин, координаты текстур
63         -0.5, -0.5, 0.0, 1.0,
64         -0.5, -0.5, 0.0, 0.0,
65         0.5, -0.5, 1.0, 1.0,
66         0.5, -0.5, 1.0, 0.0,
67     ]);
68     var n = 4; // Число вершин
69     ...
100     return n;
101 }
102
103 function initTextures(gl, n) {
104     // Создать объект текстуры
105     var texture0 = gl.createTexture(); <-(3)
106     var texture1 = gl.createTexture();
107     ...
112     // Получить ссылки на u_Sampler1 и u_Sampler2
113     var u_Sampler0 = gl.getUniformLocation(gl.program, 'u_Sampler0');
114     var u_Sampler1 = gl.getUniformLocation(gl.program, 'u_Sampler1');
115     ...
120     // Создать объекты Image
121     var image0 = new Image();
122     var image1 = new Image();
123     ...
127     // Зарегистрировать обработчик, вызываемый после загрузки изображения
128     image0.onload = function(){loadTexture(gl, n, texture0, u_Sampler0,
129                                     ➤image0, 0); };
129     image1.onload = function(){ loadTexture(gl, n, texture1, u_Sampler1,
130                                     ➤image1, 1); };
131
132     // Заставить браузер загрузить изображение
133     image0.src = '../resources/redflower.jpg';
134     image1.src = '../resources/circle.gif';
135
136     return true;
137 }
138
139 // Переменные, определяющие готовность текстурных слотов к использованию
140 var g_texUnit0 = false, g_texUnit1 = false;
141 function loadTexture(gl, n, texture, u_Sampler, image, texUnit) {
142     gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, 1); // Повернуть ось Y изображения
143     // Активировать указанный текстурный слот
144     if (texUnit == 0) {
145         gl.activeTexture(gl.TEXTURE0);
146         g_texUnit0 = true;
147     } else {
148         gl.activeTexture(gl.TEXTURE1);
149         g_texUnit1 = true;
150     }
151 }

```



```

148 // Указать тип объекта текстуры
149 gl.bindTexture(gl.TEXTURE_2D, texture);
150
151 // Определить параметры текстуры
152 gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
153 // Определить изображение текстуры
154 gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
155 // Определить указатель на текстурный слот
156 gl.uniform1i(u_Sampler, texUnit);
157 ...
161 if (g_texUnit0 && g_texUnit1) {
162     gl.drawArrays(gl.TRIANGLE_STRIP, 0, n); // Нарисовать прямоугольник
163 }
164 }

```

Сначала рассмотрим фрагментный шейдер. В программе `TexturedQuad.js` фрагментный шейдер работает с единственной текстурой, поэтому в нем объявляется единственная `uniform`-переменная `u_Sampler`. Но в этом примере используется две текстуры, поэтому мы должны определить две такие переменные, как показано ниже:

```

17 'uniform sampler2D u_Sampler0;\n' +
18 'uniform sampler2D u_Sampler1;\n' +

```

Функция `main()` фрагментного шейдера получает значения цвета для двух текселей из обеих текстур (строки 21 и 22), сохраняя их в переменных `color0` и `color1`, соответственно:

```

21 ' vec4 color0 = texture2D(u_Sampler0, v_TexCoord);\n' +
22 ' vec4 color1 = texture2D(u_Sampler1, v_TexCoord);\n' +
23 ' gl_FragColor = color0 * color1;\n' +

```

Существует множество способов вычисления значения цвета фрагмента (`gl_FragColor`) на основе нескольких текселей. В этом примере используется покомпонентное умножение, потому что результат проще поддается осмыслению. В языке GLSL ES такое умножение можно реализовать в одной строке, в виде умножения двух переменных типа `vec4`, как показано в строке 23 (см. также рис. 5.36).

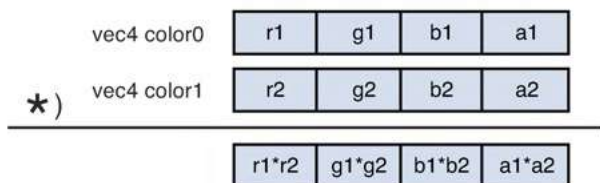


Рис. 5.36. Умножение двух переменных типа `vec4`

Несмотря на то, что в этом примере программы используется две текстуры, функция `initVertexBuffers()`, определение которой начинается в строке 60, не изменилась, в сравнении с программой `TexturedQuad.js`, потому что в ней используются одни и те же координаты текстур для обоих изображений.

В данном примере в функции `initTextures()` (строка 103) появились дополнительные строки – в ней все операции с текстурой дублируются, потому что теперь она обрабатывает два изображения текстур вместо одного, как было в предыдущем примере.

В строках 105 и 106 создаются два объекта текстур, по одному для каждой текстуры. Последние символы в именах переменных («0» в имени `texture0` и «1» в имени `texture1`) указывают, какие текстурные слоты для них будут использоваться. Это же соглашение о ссылке на номера текстурных слотов применяется к именам переменных, в которых хранятся ссылки на `uniform`-переменные (строки 113 и 114) и объекты `Image` (строки 120 и 121).

Регистрация обработчика события окончания загрузки изображения (`loadTexture()`) выполняется так же, как в программе `TexturedQuad.js`, с той лишь разницей, что его последний аргумент определяет номер текстурного слота:

```
128 image0.onload = function(){loadTexture(gl, n, texture0, u_Sampler0,
                               ➤image0, 0); };
129 image1.onload = function(){ loadTexture(gl, n, texture1, u_Sampler1,
                               ➤image1, 1); };
```

Запрос на загрузку изображений текстур выполняется в строках 131 и 132:

```
131 image0.src = '../resources/redflower.jpg';
132 image1.src = '../resources/circle.gif';
```

В этом примере программы функция `loadTexture()` изменилась – теперь она обрабатывает две текстуры. Определение функции начинается в строке 138:

```
137 var g_texUnit0 = false, g_texUnit1 = false;
138 function loadTexture(gl, n, texture, u_Sampler, image, texUnit) {
139   gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, 1); // Повернуть ось Y изображения
140   // Активировать указанный текстурный слот
141   if (texUnit == 0) {
142     gl.activeTexture(gl.TEXTURE0);
143     g_texUnit0 = true;
144   } else {
145     gl.activeTexture(gl.TEXTURE1);
146     g_texUnit1 = true;
147   }
148   // Указать тип объекта текстуры
149   gl.bindTexture(gl.TEXTURE_2D, texture);
150
151   // Определить параметры текстуры
152   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
153   // Определить изображение текстуры
154   gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
155   // Определить указатель на текстурный слот
156   gl.uniform1i(u_Sampler, texUnit);
157   ...
161   if (g_texUnit0 && g_texUnit1) {
162     gl.drawArrays(gl.TRIANGLE_STRIP, 0, n); // Нарисовать прямоугольник
163   }
```

Важная особенность функции `loadTexture()` состоит в том, что в ней нельзя заранее предсказать, какое изображение будет загружено первым, потому что загрузка выполняется браузером асинхронно. Чтобы избежать проблем, рисование в этом примере программы начинается только после загрузки обеих текстур. С этой целью в строке 137 объявляются две глобальные переменные (`g_texUnit0` и `g_texUnit1`), которые служат индикаторами готовности текстур к использованию.

Эти переменные инициализируются значением `false` в строке 137, а в инструкции `if` им присваивается значение `true` (строки со 141 по 147). Инструкция `if` проверяет значение последнего аргумента `texUnit`. Если оно равно 0, активируется текстурный слот с номером 0 и переменной `g_texUnit0` присваивается значение `true`; если оно равно 1, активируется текстурный слот с номером 1 и значение `true` присваивается переменной `g_texUnit1`.

В строке 156 номер активного текстурного слота записывается в `uniform`-переменную. Обратите внимание, что в вызов `gl.uniform1i()` передается параметр `texUnit` функции `loadTexture()`. После загрузки обеих текстур внутреннее состояние системы изменяется, как показано на рис. 5.37.

Наконец, после проверки доступности обеих текстур в строке 161, программа вызывает вершинный шейдер. Изображения объединяются и накладываются на фигуру, в результате чего формируется изображение, показанное на рис. 5.34.

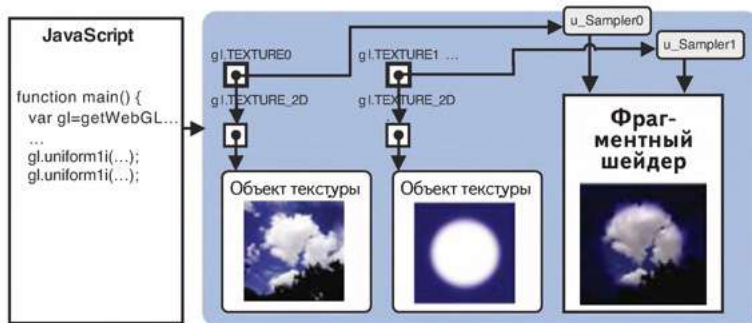


Рис. 5.37. Внутреннее состояние WebGL после загрузки двух изображений текстур

В заключение

В этой главе мы отважились погрузиться в мир WebGL. На данный момент вы приобрели все основные навыки и умения, необходимые для воспроизведения двумерных геометрических фигур с помощью WebGL и готовы сделать следующий шаг: перейти к созданию трехмерных объектов. К счастью, при работе с трехмерными объектами, шейдеры используются удивительно похожим образом, поэтому вы с успехом сможете применять все свои знания, полученные к настоящему моменту.

В оставшейся части книги основное внимание будет уделяться изучению приемов управления трехмерными объектами. Однако, прежде чем вступить в трехмерный мир, в следующей главе мы совершим краткий тур по языку шейдеров OpenGL ES (GLSL ES) и познакомимся с некоторыми его особенностями и возможностями, которые лишь вскользь упоминались в предыдущих главах.



ГЛАВА 6.

Язык шейдеров OpenGL ES (GLSL ES)

В этой главе мы временно прекратим исследование примеров WebGL-программ и познакомимся с основными особенностями OpenGL ES Shading Language (GLSL ES) – языка программирования шейдеров OpenGL.

Как вы уже знаете, шейдеры являются основным механизмом в системе WebGL, используемым при создании приложений трехмерной графики, а язык GLSL ES – это самостоятельный язык программирования шейдеров для таких приложений. Эта глава охватывает следующие темы:

- данные, переменные и типы переменных;
- векторы, матрицы, структуры, массивы и семплеры (sampler);
- операторы, инструкции управления потоком выполнения и функции;
- переменные-атрибуты, *uniform*-переменные и *varying*-переменные;
- квалификаторы точности;
- препроцессор и его директивы.

К концу этой главы вы получите хорошее понимание языка GLSL ES и приемов его использования для создания различных шейдеров. Эти знания помогут вам в реализации более сложных манипуляций с трехмерными объектами, о которых рассказывается в главах с 7 по 9. Обратите внимание, что спецификации языка могут показаться чересчур сухими, а кому-то из вас – слишком подробными. В этом случае вы можете просто пропустить данную главу и использовать ее как справочник, когда будете знакомиться с примерами в оставшейся части книги.

Краткое повторение основ шейдеров

Как показано в листингах 6.1 и 6.2, шейдеры можно конструировать точно так же, как обычные программы на языке C.

Листинг 6.1. Пример простого вершинного шейдера

```
// Вершинный шейдер
```

```
attribute vec4 a_Position;
attribute vec4 a_Color;
uniform mat4 u_MvpMatrix;
varying vec4 v_Color;
void main() {
    gl_Position = u_MvpMatrix * a_Position;
    v_Color = a_Color;
}
```

Объявления переменных помещаются в начало, а функция `main()` определяет точку входа в программу.

Листинг 6.2. Пример простого фрагментного шейдера

```
// Фрагментный шейдер
#ifdef GLSL_ES
precision mediump float;
#endif
varying vec4 v_Color;
void main() {
    gl_FragColor = v_Color;
}
```

В этой главе будет рассказываться о версии 1.00 языка GLSL ES. Однако вы должны помнить, что WebGL поддерживает не все возможности, предусмотренные спецификацией GLSL ES 1.00¹, а только часть их, необходимых в WebGL.

Обзор GLSL ES

Язык программирования **GLSL ES** был разработан на основе языка шейдеров OpenGL Shading Language (GLSL) путем упрощения и исключения некоторых функциональных возможностей, учитывая, что целевой платформой для него является бытовая электроника и встраиваемые устройства, такие как смартфоны и игровые консоли. Главная цель состояла в том, чтобы дать возможность производителям аппаратного обеспечения упрощать свои устройства, где должны выполняться программы на GLSL ES. Это дает два основных преимущества: уменьшение потребления электроэнергии устройством и, что еще важнее, снижение себестоимости.

GLSL ES поддерживает ограниченный (и, в чем-то даже расширенный) синтаксис языка C. Поэтому, если вы знакомы с языком C, вы быстро освоите язык GLSL ES. Кроме того, первоначально язык шейдеров использовался для самых разных целей, таких как реализация алгоритмов обработки изображений и математических вычислений (на графическом процессоре). Это предполагает довольно широкую область его применения, что еще больше увеличивает выгоды от изучения GLSL ES.

¹ http://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf

Привет, шейдер!

Традиционно, большинство книг по программированию начинаются с примера программы «Привет, Мир!». В нашем случае такой программе соответствует шейдер. Однако, из-за того, что в предыдущих главах мы уже видели несколько шейдеров, мы отойдем от традиции, и будем рассматривать основные особенности языка GLSL ES, используя в качестве примеров листинги 6.1 и 6.2.

Основы

Используя GLSL ES для создания шейдеров вам необходимо помнить два важных аспекта, свойственных также многим другим языкам программирования:

- имена чувствительны к регистру (имена `marina` и `Marina` считаются разными);
- каждая команда должна заканчиваться точкой с запятой (;).

Порядок выполнения

После загрузки сценария на JavaScript, браузер выполняет ее в порядке следования строк – последовательно, начиная с первой. Однако шейдеры, подобно программам на языке C, начинают выполняться с функции `main()` и потому обязательно должны иметь одну (и только одну) функцию `main()`, не имеющую параметров. Вернувшись назад, вы сможете увидеть, что оба шейдера в листингах 6.1 и 6.2 определяют единственную функцию `main()`.

Имени `main()` должно предшествовать ключевое слово `void`, указывающее, что функция не имеет возвращаемого значения. (См. раздел «Функции» далее в этой главе.) Этим GLSL ES отличается от JavaScript, где функции можно определять с помощью ключевого слова `function` и не требуется специально описывать возвращаемое значение. В GLSL ES, если функция возвращает значение, его тип необходимо указать перед именем функции, а если она ничего не возвращает, перед ее именем следует поставить ключевое слово `void`, чтобы система не ожидала получить значение.

Комментарии

Как и в программах на JavaScript, в шейдерах можно писать комментарии, используя тот же синтаксис. Ниже перечислены типы комментариев, поддерживаемые в GLSL ES:

- пара символов `//`, за которыми следует последовательность любых символов, вплоть до конца строки:

```
int kp = 496; // kp - это постоянная Капрекара
```

- пара символов `/*`, за которыми следует последовательность любых символов (включая символы перевода строки), вплоть до пары символов `*/`:

```
/* Сегодня у меня выходной.  
   Я хочу отдохнуть еще и завтра.  
*/
```

Данные (числовые и логические значения)

В языке GLSL ES поддерживается всего два типа данных.

- **Числовые значения:** GLSL ES поддерживает целые числа (например, 0, 1, 2) и вещественные числа (например, 3.14, 29.98, 0.23571). Числа без десятичной точки (.) интерпретируются как целые, а с десятичной точкой – как вещественные.
- **Логические значения:** GLSL ES поддерживает логические константы `true` и `false`.

В GLSL ES не поддерживаются строки символов, что на первый взгляд может показаться странным, но в этом есть определенный смысл, особенно если учесть, что этот язык является языком трехмерной графики.

Переменные

Как вы могли видеть в предыдущих главах, переменным можно давать любые имена, в рамках следующих простых правил:

- имена переменных могут состоять только из символов `a–z`, `A–Z`, подчеркивания (`_`) и цифр `0–9`;
- имена переменных не могут начинаться с цифр;
- ключевые слова, перечисленные в табл. 6.1, и зарезервированные слова, перечисленные в табл. 6.2 не могут использоваться в качестве имен переменных, но они могут являться частью имен, например, использование имени `if` для переменной будет считаться ошибкой, а имя `iffy` – нет;
- имена переменных, начинающиеся с префиксов `gl_`, `webgl_` и `webgl_` зарезервированы для использования в реализации OpenGL ES; имена пользовательских переменных не должны начинаться с этих префиксов.

Таблица 6.1. Ключевые слова в языке GLSL ES

<code>attribute</code>	<code>bool</code>	<code>break</code>	<code>bvec2</code>	<code>bvec3</code>	<code>bvec4</code>
<code>const</code>	<code>continue</code>	<code>discard</code>	<code>do</code>	<code>else</code>	<code>false</code>
<code>float</code>	<code>for</code>	<code>highp</code>	<code>if</code>	<code>in</code>	<code>inout</code>
<code>int</code>	<code>invariant</code>	<code>ivec2</code>	<code>ivec3</code>	<code>ivec4</code>	<code>lowp</code>
<code>mat2</code>	<code>mat3</code>	<code>mat4</code>	<code>mediump</code>	<code>out</code>	<code>precision</code>
<code>return</code>	<code>sampler2D</code>	<code>samplerCube</code>	<code>struct</code>	<code>true</code>	<code>uniform</code>
<code>varying</code>	<code>vec2</code>	<code>vec3</code>	<code>vec4</code>	<code>void</code>	<code>while</code>

Таблица 6.2. Зарезервированные слова для будущих версий GLSL ES

<code>asm</code>	<code>cast</code>	<code>class</code>	<code>default</code>
<code>double</code>	<code>dvec2</code>	<code>dvec3</code>	<code>dvec4</code>
<code>enum</code>	<code>extern</code>	<code>external</code>	<code>fixed</code>
<code>flat</code>	<code>fvec2</code>	<code>fvec3</code>	<code>fvec4</code>
<code>goto</code>	<code>half</code>	<code>hvec2</code>	<code>hvec3</code>
<code>hvec4</code>	<code>inline</code>	<code>input</code>	<code>interface</code>
<code>long</code>	<code>namespace</code>	<code>noinline</code>	<code>output</code>
<code>packed</code>	<code>public</code>	<code>sampler1D</code>	<code>sampler1DShadow</code>
<code>sampler2DRect</code>	<code>sampler2DRectShadow</code>	<code>sampler2DShadow</code>	<code>sampler3D</code>
<code>sampler3DRect</code>	<code>short</code>	<code>sizeof</code>	<code>static</code>
<code>superp</code>	<code>switch</code>	<code>template</code>	<code>this</code>
<code>typedef</code>	<code>union</code>	<code>unsigned</code>	<code>using</code>
<code>volatile</code>			

GLSL ES – типизированный язык

GLSL ES не требует использовать ключевое слово `var` для объявления переменных, но он требует указывать типы данных, которые они могут хранить. Как вы уже знаете по примерам программ, синтаксис объявления переменных имеет вид:

```
<тип данных> <имя переменной>
```

например:

```
vec4 a_Position
```

Как уже говорилось выше, определяя функции, такие как `main()`, также требуется указывать типы возвращаемых ими значений. Точно так же тип переменной слева и тип значения справа от оператора присваивания (`=`) должны совпадать; несовпадение типов будет рассматриваться как ошибка.

По этим причинам GLSL ES называют типизированным языком, в том смысле, что он принадлежит к классу языков, требующих указывать типы и следить за ними.

Простые типы

В табл. 6.3 перечислены простые типы, поддерживаемые в языке GLSL ES.

Таблица 6.3. Простые типы в языке GLSL

Тип	Описание
float	Тип данных, представляющий одно вещественное число. Указывает, что переменная этого типа будет хранить единственное вещественное число.
int	Тип данных, представляющий одно целое число. Указывает, что переменная этого типа будет хранить единственное целое число.
bool	Тип данных, представляющий логическое значение. Указывает, что переменная этого типа будет хранить логическое значение.

Указывая типы переменных, мы даем возможность системе WebGL проверить наличие ошибок в программе и увеличиваем ее эффективность. Ниже приводятся примеры объявлений переменных простых типов.

```
float klimt; // Переменная будет хранить единственное вещественное число
int utrillo; // Переменная будет хранить единственное целое число
bool doga; // Переменная будет хранить единственное логическое значение
```

Присваивание и преобразования типов

Присваивание значений переменным выполняется с помощью оператора присваивания (=). Как упоминалось выше, из-за того, что GLSL ES является типизированным языком, если тип переменной слева не совпадает с типом значения справа, это приводит к ошибке:

```
int i = 8; // ОК
float f1 = 8; // Ошибка
float f2 = 8.0; // ОК
float f3 = 8.0f; // Ошибка: выражения, такие как 8.0f, используемые в C,
// в языке GLSL ES недопустимы.
```

Семантически, 8 и 8.0 – это одно и то же значение. Однако, попытка присвоить 8 вещественной переменной f1 приводит к ошибке. В данном случае появится следующее сообщение об ошибке:

```
failed to compile shader: ERROR: 0:11: '=' : cannot convert
from 'const mediump int' to 'float'.
(ошибка компиляции шейдера: ОШИБКА: 0:11: '=' : не может преобразовать
тип 'const mediump int' в тип 'float'.)
```

Если потребуется присвоить целочисленное значение переменной вещественного типа, это значение необходимо будет преобразовать в вещественное число. Такое преобразование называется **приведением типа**. Чтобы преобразовать целое число в вещественное, можно воспользоваться встроенной функцией `float()`:

```
int i = 8;
float f1 = float(i); // преобразует 8 в 8.0 и присвоит переменной f1
float f2 = float(8); // эквивалентная операция
```

GLSL ES поддерживает еще несколько встроенных функций для приведения типов, которые перечислены в табл. 6.4.

Таблица 6.4. Встроенные функции для приведения типов

Преобразование	Функция	Описание
В целое число	<code>int(float)</code>	Дробная часть вещественного числа отбрасывается (например, $3.14 \rightarrow 3$).
	<code>int(bool)</code>	Значение <code>true</code> преобразуется в <code>1</code> , значение <code>false</code> – в <code>0</code> .
В вещественное число	<code>float(int)</code>	Целое число преобразуется в вещественное (например, $8 \rightarrow 8.0$).
	<code>float(bool)</code>	Значение <code>true</code> преобразуется в <code>1.0</code> , значение <code>false</code> – в <code>0.0</code> .
В логическое значение	<code>bool(int)</code>	<code>0</code> преобразуется в <code>false</code> , любое ненулевое значение – в <code>true</code> .
	<code>bool(float)</code>	<code>0.0</code> преобразуется в <code>false</code> , любое ненулевое значение – в <code>true</code> .

Операции

Операторы, применимые к простым типам, похожи на соответствующие им операторы в JavaScript. Все они перечислены в табл. 6.5.

Таблица 6.5. Операторы, применимые к простым типам

Оператор	Операция	Поддерживаемые типы данных
<code>-</code>	Отрицание (например, для обозначения отрицательного числа)	<code>int</code> или <code>float</code>
<code>*</code>	Умножение	<code>int</code> или <code>float</code> . Тип данных результата совпадает с типом данных операндов.
<code>/</code>	Деление	
<code>+</code>	Сложение	
<code>-</code>	Вычитание	
<code>++</code>	Инкремент (префиксный и постфиксный)	<code>int</code> или <code>float</code> . Тип данных результата совпадает с типом данных операнда.
<code>--</code>	Декремент (префиксный и постфиксный)	
<code>=</code>	Присваивание	<code>int</code> , <code>float</code> или <code>bool</code> .
<code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code>	Комбинированные операции присваивания	<code>int</code> или <code>float</code>
<code><</code> <code>></code> <code><=</code> <code>>=</code>	Сравнение	<code>int</code> или <code>float</code>

Оператор	Операция	Поддерживаемые типы данных
<code>== !=</code>	Сравнение (проверка на равенство)	<code>int</code> , <code>float</code> или <code>bool</code> .
<code>!</code>	Логическое отрицание	значение типа <code>bool</code> или выражение, возвращающее значение типа <code>bool</code> . ^[1]
<code>&&</code>	Логическое «И»	
<code> </code>	Логическое «ИЛИ»	
<code>^^</code>	Логическое «Исключающее ИЛИ» ^[2]	
<code>condition? expression1:expression2</code>	Тернарный выбор	<code>condition</code> – значение типа <code>bool</code> или выражение, возвращающее значение типа <code>bool</code> . Выражения <code>expression1</code> и <code>expression2</code> могут иметь любой тип, кроме массива.

[1] Второй операнд в операции логического «И» (`&&`) вычисляется, только если первый операнд имеет значение `true`. Второй операнд в операции логического «ИЛИ» (`||`) вычисляется, только если первый операнд имеет значение `false`.

[2] Результат операции получит значение `true`, только если один из операндов (левый или правый) имеет значение `true`. Если оба операнда имеют значение `true`, в результате получится `false`.

Ниже приводятся примеры простых операций:

```
int i1 = 954, i2 = 459;
int kp = i1 - i2;           // переменной kp будет присвоено 495
float f = float(kp) + 5.5; // переменной f будет присвоено 500.5
```

Векторы и матрицы

В языке GLSL ES поддерживаются типы данных для представления векторов и матриц, которые, как вы уже могли убедиться, весьма удобны для выполнения операций с компьютерной графикой. Значения обоих типов состоят из множества элементов. Вектор хранит данные в виде списка и его удобно использовать для представления координат вершины или цвета. Матрица хранит данные в виде массива и ее удобно использовать для представления матриц преобразований. На рис. 6.1 представлены примеры данных обоих типов.

$$(3 \ 7 \ 1) \quad \begin{bmatrix} 3 & 7 & 1 \\ 1 & 5 & 3 \\ 4 & 0 & 7 \end{bmatrix}$$

Рис. 6.1.
Вектор и матрица

В GLSL ES поддерживается несколько разновидностей векторов и матриц, которые перечислены в табл. 6.6.

Таблица 6.6. Типы векторов и матриц

Категория	Типы в GLSL ES	Описание
Векторы	<code>vec2, vec3, vec4</code>	Типы данных для представления векторов, состоящих из 2, 3 и 4 вещественных чисел.
	<code>ivec2, ivec3, ivec4</code>	Типы данных для представления векторов, состоящих из 2, 3 и 4 целых чисел.
	<code>bvec2, bvec3, bvec4</code>	Типы данных для представления векторов, состоящих из 2, 3 и 4 логических значений.
Матрицы	<code>mat2, mat3, mat4</code>	Типы данных для представления матриц 2×2, 3×3 и 4×4, состоящих из вещественных чисел (4, 9 и 16, соответственно).

Ниже демонстрируется несколько примеров использования векторов и матриц:

```
vec3 position; // переменная-вектор с тремя вещественными компонентами
                // Например: (10.0, 20.0, 30.0)
ivec2 offset;  // переменная-вектор с двумя целочисленными компонентами
                // Например: (10, 20)
mat4 mvpMatrix; // переменная-матрица 4×4 с вещественными числами
```

Присваивание и конструирование

Присваивание данных переменным-векторам и переменным-матрицам выполняется с помощью оператора `=`. Не забывайте, что тип переменной слева от оператора присваивания и тип данных/переменных справа должны совпадать. Кроме того, число элементов в переменной-векторе или переменной-матрице слева должно быть равно числу элементов данных справа. Для иллюстрации этого ниже приводится пример, который генерирует ошибку:

```
vec4 position = 1.0; // переменная типа vec4 требует четыре вещественных числа
```

В данном случае, из-за того, что переменная типа `vec4` требует четыре вещественных числа, вы должны передать их каким-то образом. Решить эту задачу можно с помощью встроенных функций, имена которых совпадают с именами типов данных; например, с переменной типа `vec4` можно использовать конструктор `vec4()`. (См. главу 2, «Первые шаги в WebGL».) Так, чтобы присвоить числа 1.0, 2.0, 3.0 и 4.0 переменной типа `vec4`, можно с помощью `vec4()` связать их в единый вектор, как показано ниже:

```
vec4 position = vec4(1.0, 2.0, 3.0, 4.0);
```

Функции, создающие значения определенного типа, называются **функциями-конструкторами**, а имена конструкторов всегда совпадают с именами типов данных.

Конструкторы векторов

Векторы играют критически важную роль в GLSL ES, поэтому, как можно догадаться, существует масса способов определения аргументов при вызове конструктора вектора. Например:

```
vec3 v3 = vec3(1.0, 0.0, 0.5); // присвоит переменной v3 вектор (1.0, 0.0, 0.5)
vec2 v2 = vec2(v3); // присвоит переменной v2 вектор (1.0, 0.0),
// используя 1-й и 2-й элементы вектора v3
vec4 v4 = vec4(1.0); // присвоит переменной v4 вектор (1.0, 1.0, 1.0, 1.0)
```

Во втором примере конструктор игнорирует третий элемент вектора `v3` и для создания нового вектора использует только первый и второй элементы. Аналогично, в третьем примере, если в вызов конструктора вектора передать единственное значение, оно будет использовано для инициализации всех элементов нового вектора. Однако, если конструктору передать больше одного значения, но меньше, чем число элементов в конструируемом векторе, это приведет к ошибке.

Наконец, вектор можно сконструировать из нескольких векторов:

```
vec4 v4b = vec4(v2, v4); // присвоит переменной v4b вектор (1.0, 0.0, 1.0, 1.0)
```

Здесь действует следующее правило: первые элементы нового вектора получают значения элементов из вектора `v2`, а последующие – значения элементов из вектора `v4`.

Конструкторы матриц

Для создания матриц тоже имеются свои конструкторы, которые действуют подобно конструкторам векторов. Однако вы должны помнить, что элементы должны следовать в порядке расположения по столбцам (этот порядок расположения изображен на рис. 3.27). Примеры ниже демонстрируют некоторые особенности использования конструкторов матриц:

- если конструктору передать несколько значений, он создаст из них матрицу, предполагая, что аргументы следуют по столбцам:

```
mat4 m4 = mat4 ( 1.0, 2.0, 3.0, 4.0,
                 5.0, 6.0, 7.0, 8.0,
                 9.0, 10.0, 11.0, 12.0,
                 13.0, 14.0, 15.0, 16.0 );
```

1.0	5.0	9.0	13.0
2.0	6.0	10.0	14.0
3.0	7.0	11.0	15.0
4.0	8.0	12.0	16.0

- если конструктору передать несколько векторов, он создаст матрицу, предполагая, что элементы векторов представляют содержимое матрицы в порядке следования по столбцам:

```
// конструирование матрицы типа mat2 из двух векторов типа vec2
vec2 v2_1 = vec2(1.0, 3.0);
vec2 v2_2 = vec2(2.0, 4.0);
mat2 m2_1 = mat2(v2_1, v2_2); // 1.0 2.0
// 3.0 4.0

// конструирование матрицы типа mat2 из вектора типа vec4
```

```
vec4 v4 = vec4(1.0, 3.0, 2.0, 4.0);
mat2 m2_2 = mat2(v4); // 1.0 2.0
                // 3.4 4.0
```

- если конструктору передать несколько отдельных значений и векторов, он создаст матрицу, предполагая, что эти значения и элементы векторов представляют содержимое матрицы в порядке следования по столбцам:

```
// конструирование матрицы типа mat2
// из двух вещественных чисел и вектора типа vec2
mat2 m2 = mat2(1.0, 3.0, v2_2); // 1.0 2.0
                                // 3.0 4.0
```

- если конструктору передать единственное значение, он создаст матрицу, используя это значение для инициализации элементов диагонали:

```
mat4 m4 = mat4(1.0); // 1.0 0.0 0.0 0.0
                    // 0.0 1.0 0.0 0.0
                    // 0.0 0.0 1.0 0.0
                    // 0.0 0.0 0.0 1.0
```

Так же, как и в случае с конструкторами векторов, если конструктору матрицы передать недостаточное число значений (но больше одного), это приведет к ошибке.

```
mat4 m4 = mat4(1.0, 2.0, 3.0); // Ошибка. Для mat4 требуется 16 элементов
```

Доступ к компонентам

Доступ к компонентам вектора или матрицы можно получить с помощью операторов `.` и `[]`, как показано в следующих разделах.

Оператор точки (.)

Доступ к отдельному компоненту вектора можно получить, используя имя переменной, следующий за ним оператор точки `.` и затем имя компонента, как показано в табл. 6.7.

Таблица 6.7. Имена компонентов

Категория	Описание
<code>x, y, z, w</code>	Удобно использовать для доступа к координатам вершины.
<code>r, g, b, a</code>	Удобно использовать для доступа к значению цвета.
<code>s, t, p, q</code>	Удобно использовать для доступа к текстурным координатам. (Обратите внимание, что в этой книге используются только координаты <code>s</code> и <code>t</code> . Координата <code>R</code> обозначается символом <code>p</code> , потому что символ <code>r</code> уже используется для обозначения красной составляющей цвета.)

Так как векторы используются для хранения разнотипных данных, таких как координаты вершин, составляющие цвета и координаты текстур, поддерживается

три типа имен компонентов для увеличения удобочитаемости программ. Но, любое имя — *x*, *r* или *s* — обеспечивает доступ к первому компоненту вектора; любое имя — *y*, *g* или *t* — обеспечивает доступ ко второму компоненту вектора; и так далее, поэтому их можно использовать взаимозаменяемо. Например:

```
vec3 v3 = vec3(1.0, 2.0, 3.0); // присвоит переменной v3 вектор (1.0, 2.0, 3.0)
float f;

f = v3.x; // присвоит переменной f значение 1.0
f = v3.y; // присвоит переменной f значение 2.0
f = v3.z; // присвоит переменной f значение 3.0

f = v3.r; // присвоит переменной f значение 1.0
f = v3.s; // присвоит переменной f значение 1.0
```

Как следует из комментариев в этих примерах, имена *x*, *r* и *s* являются разными, но все они ссылаются на первый компонент вектора. Попытка обратиться к компоненту за пределами вектора приведет к ошибке:

```
f = v3.w; // w обеспечивает доступ к четвертому элементу,
           // который отсутствует в данном векторе.
```

Имеется возможность выбрать сразу несколько компонентов, комбинируя их имена (из одной категории после точки (.)). Этот прием называют **смешиванием** (swizzling). В следующем примере используются имена *x*, *y*, *z* и *w*, но с тем же успехом можно использовать другие категории имен:

```
vec2 v2;
v2 = v3.xy; // присвоит вектор (1.0, 2.0)
v2 = v3.yz; // присвоит вектор (2.0, 3.0). Любой компонент можно опустить
v2 = v3.xz; // присвоит вектор (1.0, 3.0). Любой компонент можно пропустить
v2 = v3.yx; // присвоит вектор (2.0, 1.0). Компоненты могут следовать в любом порядке
v2 = v3.xx; // присвоит вектор (1.0, 1.0). Компоненты можно повторять

vec3 v3a;
v3a = v3.zyx; // присвоит вектор (3.0, 2.0, 1.0). Можно использовать все имена
```

Имена компонентов можно также использовать слева от оператора присваивания (=):

```
vec4 position = vec4(1.0, 2.0, 3.0, 4.0);
position.xw = vec2(5.0, 6.0); // position = (5.0, 2.0, 3.0, 6.0)
```

Запомните, что комбинировать можно только имена из одной категории, например, выражение *v3.was* является недопустимым.

Оператор []

Помимо оператора точки (.), обращаться к компонентам векторов и матриц можно с помощью оператора индексирования массивов []. Обратите внимание, что чтение элементов матриц так же выполняется по столбцам. Как и в JavaS-

`cript`, нумерация элементов начинается с 0, поэтому применение `[0]` к матрице вернет ее первый столбец, `[1]` вернет второй столбец, `[2]` – третий, и так далее. Например:

```
mat4 m4 = mat4 ( 1.0,  2.0,  3.0,  4.0,
                 5.0,  6.0,  7.0,  8.0,
                 9.0, 10.0, 11.0, 12.0,
                 13.0, 14.0, 15.0, 16.0);
vec4 v4 = m4[0]; // Вернет 1-й столбец из m4: (1.0, 2.0, 3.0, 4.0)
```

Для выбора единственного элемента матрицы по номеру столбца и строки можно использовать два оператора `[]`:

```
float m23 = m4[1][2]; // присвоит переменной m23 третий компонент
                      // второго столбца матрицы m4 (7.0).
```

Также для выбора отдельных компонентов, в сочетании с оператором `[]`, можно использовать их имена, как показано ниже:

```
float m32 = m4[2].y; // присвоит переменной m32 второй компонент
                     // третьего столбца матрицы m4 (10.0).
```

Единственное ограничение оператора `[]` – он допускает использовать только числовые константы. Индекс можно определить, как:

- целочисленный литерал (например, 0 или 1);
- глобальную или локальную переменную, объявленную с квалификатором `const`, за исключением параметров функций (см. раздел «Квалификатор `const`»);
- индекс цикла (см. раздел «Условные операторы и циклы»);
- выражения, составленные из любых элементов, перечисленных выше.

В следующем примере в качестве индекса используется переменная типа `int`, объявленная с квалификатором `const`:

```
const int index = 0; // ключевое слово "const" указывает,
                    // что переменная доступна только для чтения.
vec4 v4a = m4[index]; // то же, что и m4[0]
```

В следующем примере в качестве индекса используется выражение, составленное из константы и числового литерала:

```
vec4 v4b = m4[index + 1]; // то же, что и m4[1]
```

Помните, что в качестве индекса нельзя использовать переменную типа `int`, объявленную без квалификатора `const`, потому что она не является константой (исключение составляет переменная цикла):

```
int index2 = 0;
vec4 v4c = m4[index2]; // Ошибка: потому что index2 не является константой.
```

Операции

К векторам и матрицам можно применять операторы, перечисленные в табл. 6.8. Эти операторы похожи на операторы для работы с простыми типами. Обратите внимание, что из операторов сравнения к матрицам и векторам могут применяться только операторы «равно» (`==`) и «не равно» (`!=`). Операторы `<`, `>`, `<=` и `>=` нельзя использовать для сравнения векторов и матриц. Для выяснения отношений вида меньше/больше между матрицами или векторами следует использовать встроенные функции, такие как `lessThan()`. (См. приложение В, «Встроенные функции GLSL ES 1.0».)

Таблица 6.8. Операторы для работы с векторами и матрицами

Оператор	Операция	Применимость к типам данных
<code>*</code>	Умножение	<code>vec[234]</code> и <code>mat[234]</code> . Операции с типами <code>vec[234]</code> и <code>mat[234]</code> будут описаны ниже. Тип результата определяется типом операндов.
<code>/</code>	Деление	
<code>+</code>	Сложение	
<code>-</code>	Вычитание	
<code>++</code>	Инкремент (префиксный и постфиксный)	<code>vec[234]</code> и <code>mat[234]</code> . Тип результата определяется типом операнда.
<code>--</code>	Декремент (префиксный и постфиксный)	
<code>=</code>	Присваивание	<code>vec[234]</code> и <code>mat[234]</code> .
<code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code>	Арифметическая операция с присваиванием	<code>vec[234]</code> и <code>mat[234]</code> .
<code>==</code> , <code>!=</code>	Сравнение	<code>vec[234]</code> и <code>mat[234]</code> . Оператор <code>==</code> возвращает <code>true</code> , только если все компоненты сравниваемых операндов равны. Оператор <code>!=</code> возвращает <code>true</code> , если операнды отличаются хотя бы одним компонентом [1].

[1] Получить результаты сравнений отдельных пар компонентов можно с помощью встроенных функций `equal()` и `notEqual()`. (См. приложение В.)

Обратите внимание, что при применении арифметических операторов к векторам или матрицам, они действуют независимо, отдельно для каждой пары компонентов векторов или матриц.

Примеры

Далее демонстрируются наиболее часто встречающиеся случаи применения операторов. В этих примерах предполагается, что переменные объявлены, как показано ниже:

```
vec3 v3a, v3b, v3c;
mat3 m3a, m3b, m3c;
float f;
```

Операции с векторами и вещественными числами

Следующий пример демонстрирует использование оператора +:

```
// Следующий пример демонстрирует использование оператора +,  
// однако операторы -, * и / применяются аналогично.  
v3b = v3a + f; // v3b.x = v3a.x + f;  
           // v3b.y = v3a.y + f;  
           // v3b.z = v3a.z + f;
```

Например, для $v3a = \text{vec3}(1.0, 2.0, 3.0)$ и $f = 1.0$ в результате получится $v3b = (2.0, 3.0, 4.0)$.

Операции с векторами

Эти операторы применяются к каждому компоненту вектора:

```
// Следующий пример демонстрирует использование оператора +,  
// однако операторы -, * и / применяются аналогично.  
v3c = v3a + v3b; // v3a.x + v3b.x;  
           // v3a.y + v3b.y;  
           // v3a.z + v3b.z;
```

Например, для $v3a = \text{vec3}(1.0, 2.0, 3.0)$ и $v3b = \text{vec3}(4.0, 5.0, 6.0)$ в результате получится $v3c = (5.0, 7.0, 9.0)$.

Операции с матрицами и вещественными числами

Эти операторы применяются к каждому компоненту матрицы:

```
// Следующий пример демонстрирует использование оператора +,  
// однако операторы -, * и / применяются аналогично.  
m3b = m3a * f; // m3b[0].x = m3a[0].x * f; m3b[0].y = m3a[0].y * f;  
           // m3b[0].z = m3a[0].z * f;  
           // m3b[1].x = m3a[1].x * f; m3b[1].y = m3a[1].y * f;  
           // m3b[1].z = m3a[1].z * f;  
           // m3b[2].x = m3a[2].x * f; m3b[2].y = m3a[2].y * f;  
           // m3b[2].z = m3a[2].z * f;
```

Умножение матрицы на вектор

Результатом умножения матрицы на вектор является сумма произведений каждого элемента матрицы на вектор. Порядок вычислений определяется формулой 3.5, которая приводится в главе 3, «Рисование и преобразование треугольников»:

```
v3b = m3a * v3a; // v3b.x = m3a[0].x * v3a.x + m3a[1].x * v3a.y  
           //           + m3a[2].x * v3a.z;  
           // v3b.y = m3a[0].y * v3a.x + m3a[1].y * v3a.y  
           //           + m3a[2].y * v3a.z;  
           // v3b.z = m3a[0].z * v3a.x + m3a[1].z * v3a.y  
           //           + m3a[2].z * v3a.z;
```

Умножение вектора на матрицу

Умножение вектора на матрицу так же возможно, как показывает следующий пример. Обратите внимание, что этот результат отличается от результата умножения матрицы на вектор:

```
v3b = v3a * m3a; // v3b.x = v3a.x * m3a[0].x + v3a.y * m3a[0].y
//              + v3a.z * m3a[0].z;
// v3b.y = v3a.x * m3a[1].x + v3a.y * m3a[1].y
//              + v3a.z * m3a[1].z;
// v3b.z = v3a.x * m3a[2].x + v3a.y * m3a[2].y
//              + v3a.z * m3a[2].z;
```

Умножение матриц

Порядок умножения матриц определяется формулой 4.4, которая приводится в главе 4, «Дополнительные преобразования и простая анимация»:

```
m3c = m3a * m3b; // m3c[0].x = m3a[0].x * m3b[0].x + m3a[1].x * m3b[0].y
//              + m3a[2].x * m3b[0].z;
// m3c[1].x = m3a[0].x * m3b[1].x + m3a[1].x * m3b[1].y
//              + m3a[2].x * m3b[1].z;
// m3c[2].x = m3a[0].x * m3b[2].x + m3a[1].x * m3b[2].y
//              + m3a[2].x * m3b[2].z;

// m3c[0].y = m3a[0].y * m3b[0].x + m3a[1].y * m3b[0].y
//              + m3a[2].y * m3b[0].z;
// m3c[1].y = m3a[0].y * m3b[1].x + m3a[1].y * m3b[1].y
//              + m3a[2].y * m3b[1].z;
// m3c[2].y = m3a[0].y * m3b[2].x + m3a[1].y * m3b[2].y
//              + m3a[2].y * m3b[2].z;

// m3c[0].z = m3a[0].z * m3b[0].x + m3a[1].z * m3b[0].y
//              + m3a[2].z * m3b[0].z;
// m3c[1].z = m3a[0].z * m3b[1].x + m3a[1].z * m3b[1].y
//              + m3a[2].z * m3b[1].z;
// m3c[2].z = m3a[0].z * m3b[2].x + m3a[1].z * m3b[2].y
//              + m3a[2].z * m3b[2].z;
```

Структуры

В языке GLSL ES поддерживается также возможность определения пользовательских типов, называемых **структурами**, путем объединения с помощью ключевого слова `struct` других, уже определенных типов. Например:

```
struct light { // определение структуры с именем "light"
    vec4 color;
    vec3 position;
}
light l1, l2; // объявление переменных "l1" и "l2" типа "light"
```

В этом примере определяется новый структурный тип `light`, включающий два поля: переменные `color` и `position`. Затем объявляются две переменные — `l1` и

12 – типа `light`. В отличие от языка C, в GLSL ES не требуется использовать ключевое слово `typedef`, потому что по умолчанию имя структуры интерпретируется как имя типа.

Кроме того, переменные нового типа можно объявлять непосредственно в определении структуры:

```
struct light {    // определение структуры и объявление переменных
    vec4 color;   // цвет подсветки
    vec3 position; // позиция источника света
} l1;            // переменная "l1" типа "light"
```

Присваивание и конструирование

Структуры поддерживают стандартные конструкторы с именами, совпадающими с именами структур. Порядок следования аргументов в вызове конструктора и их типы должны соответствовать определению структуры. На рис. 6.2 приводится пример использования конструктора.

```
l1 = light(vec4(0.0, 1.0, 0.0, 1.0), vec3(8.0, 3.0, 0.0));
           color           position
```

Рис. 6.2. Конструктор структуры

Доступ к членам

Обратиться к любому члену структуры можно, добавив к имени переменной точку (.) и имя члена. Например:

```
vec4 color = l1.color;
vec3 position = l1.position;
```

Операции

К каждому члену структуры можно применять любые операторы, допустимые для данных этого типа. Однако, к самим структурам можно применять только операторы присваивания (=) и сравнения (== и !=); см. табл. 6.9.

Таблица 6.9. Операторы для работы со структурами

Оператор	Операция	Применимость к типам данных
=	Присваивание	Операторы присваивания и сравнения нельзя применять к структурам, содержащим члены-массивы или члены-семплы.
==, !=	Сравнение	

Оператор `==` возвращает `true`, только если все члены сравниваемых структур равны. Оператор `!=` возвращает `true`, если операнды отличаются значением хотя бы одного члена.

Массивы

Массивы в языке GLSL ES имеют много общего с массивами в JavaScript, с той лишь разницей, что в GLSL ES поддерживаются только одномерные массивы. В отличие от JavaScript, для создания массива в GLSL ES не требуется использовать оператор `new`, и массивы не имеют методов, таких как `push()` и `pop()`. Массивы можно объявлять указанием имени, за которым следуют квадратные скобки (`[]`), заключающие размер массива. Например:

```
float floatArray[4]; // объявление массива четырех вещественных чисел
vec4 vec4Array[2];  // объявление массива двух векторов vec4
```

Размер массива должен определяться **целочисленным константным выражением** и иметь величину больше нуля. Целочисленное константное выражение определяется, как:

- целочисленный литерал (например, 0 или 1);
- глобальная или локальная переменная, объявленная с квалификатором `const`, за исключением параметров функций (см. раздел «Квалификатор `const`»);
- выражения, составленные из любых элементов, перечисленных выше.

То есть, в следующем примере будет сгенерирована ошибка:

```
int size = 4;
vec4 vec4Array[size]; // Ошибка. Эта ошибка не возникла бы, если бы
                      // переменная была объявлена, как "const int size = 4;"
```

Обратите внимание, что массив нельзя объявить с квалификатором `const`.

Доступ к отдельным элементам массивов осуществляется с применением оператора индексирования (`[]`). Отметим, что как и в языке C, нумерация индексов начинается с 0. Например, третий элемент массива вещественных чисел `floatArray`, объявленного выше, доступен, как показано ниже:

```
float f = floatArray[2];
```

В качестве индексов могут использоваться только целочисленные константные выражения и `uniform`-переменные (см. раздел «`uniform`-переменные»). Кроме того, в отличие от JavaScript или C, массивы в GLSL ES нельзя инициализировать при объявлении. То есть, каждый элемент массива должен быть инициализирован явно, например:

```
vec4Array[0] = vec4(4.0, 3.0, 6.0, 1.0);
vec4Array[1] = vec4(3.0, 2.0, 0.0, 1.0);
```

Массивы поддерживают только оператор `[]`. Однако элементы массива поддерживают все стандартные операторы, доступные для данного типа. Например, к элементам `floatArray` или `vec4Array` можно применить оператор умножения:

```
// умножить второй элемент floatArray на 3.14
float f = floatArray[1] * 3.14;
```

```
// умножить первый элемент vec4Array на vec4(1.0, 2.0, 3.0, 4.0);
vec4 v4 = vec4Array[0] * vec4(1.0, 2.0, 3.0, 4.0);
```

Семплы

В языке GLSL ES поддерживаются специальные типы данных для доступа к текстурам, которые называются семплами (sampler). (См. главу 5, «Цвет и текстура».) Всего доступна два типа семплов: `sampler2D` и `samplerCube`. Переменными-семплами могут быть только `uniform`-переменные (см. раздел «`uniform`-переменные») или аргументы функций (таких как `texture2D()`), используемые для ссылки на текстуры. (См. приложение В.) Например:

```
uniform sampler2D u_Sampler;
```

Кроме того, единственное значение, которое можно присвоить такой переменной, — это номер текстурного слота, и с этой целью должен использоваться метод `WebGL gl.uniform1i()`. Например, в программе `TexturedQuad.js` (глава 5) для передачи текстурного слота с номером 0 в шейдер производится вызов `gl.uniform1i(u_Sampler, 0)`.

Переменные-семплы не могут быть операндами ни в каких операциях, кроме присваивания (=) и сравнения (== и !=).

В отличие от других типов, описанных в предыдущих разделах, число переменных-семплов ограничено, в зависимости от типа шейдера (см. табл. 6.10). Ключевое слово `mediump`, в табл. 6.10, — это квалификатор точности. (Подробнее об этом квалификаторе рассказывается в разделе «Квалификаторы точности», ближе к концу главы.)

Таблица 6.10. Минимальное число переменных-семплов

Шейдеры	Встроенные константы, представляющие максимальное число	Минимальное число
Вершинный шейдер	<code>const mediump int gl_MaxVertexTextureImageUnits</code>	0
Фрагментный шейдер	<code>const mediump int gl_MaxTextureImageUnits</code>	8

Приоритеты операторов

В табл. 6.11 указаны приоритеты операторов. Обратите внимание, что в таблице присутствует несколько операторов, которые не описываются в этой книге. Мы включили их для справки.

Таблица 6.11. Приоритеты операторов

Приоритет	Операторы
1	Группирующие скобки ().
2	Вызовы функций, конструкторов (), индексирование массивов [], точка ().

Приоритет	Операторы
3	Инкремент/декремент (++), унарный минус (-), инверсия (~) , логическое отрицание (!)
4	Умножение (*), деление (/), остаток от деления (%)
5	Сложение (+), вычитание (-).
6	Поразрядный сдвиг (<<, >>).
7	Операторы сравнения (<, <=, >=, >).
8	Операторы проверки равенства/неравенства (==, !=).
9	Поразрядное «И» (&).
10	Поразрядное «Исключающее ИЛИ» (^).
11	Поразрядное «ИЛИ» ().
12	Логическое «И» (&&).
13	Логическое «Исключающее ИЛИ» (^ ^).
14	Логическое «ИЛИ» ().
15	Тернарный оператор выбора (?:).
16	Присваивание (=), комбинированное присваивание (+=, -=, /=, %=, <<=, >>=, &=, ^=, =)
17	Последовательность (,)

Жирным выделены операторы, зарезервированные для реализации в будущих версиях GLSL.

Условные операторы и циклы

В языке шейдеров используются практически те же условные операторы и операторы циклов, что и в языках JavaScript и C.

Инструкции *if* и *if-else*

Управление потоком выполнения можно осуществлять с помощью инструкций *if* и *if-else*. Инструкция *if-else* используется следующим образом:

```
if ( условное-выражение1 ) {
    команды, выполняемые, если условное-выражение1 истинно.
} else if ( условное-выражение2 ) {
    команды, выполняемые, если условное-выражение1 ложно,
    а условное-выражение2 истинно.
} else {
    команды, выполняемые, если условное-выражение1 и условное-выражение2 ложны.
}
```

Ниже показан фрагмент кода, использующий инструкцию *if-else*:

```
if(distance < 0.5) {
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0); // красный
```



```
} else {  
    gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0); // зеленый  
}
```

Как следует из этого примера, условное выражение в инструкции `if` или `if-else` должно иметь логическое значение или преобразовываться в логическое значение. Вектор логических значений, например типа `bvec2`, нельзя использовать в качестве условного выражения.

Инструкция `switch` не поддерживается, а также имейте в виду, что применение инструкций `if` и `if-else` в шейдерах замедляет скорость их выполнения.

Инструкция `for`

Инструкция `for` используется, как показано ниже:

```
for(инструкция-инициализации; условное-выражение; выражение-изменения-индекса)  
{  
    команды, выполняемые в цикле.  
}
```

Например:

```
for (int i = 0; i < 3; i++) {  
    sum += i;  
}
```

Обратите внимание, что индекс цикла `for` (`i` в предыдущем примере) должен объявляться непосредственно в инструкции-инициализации. Условное-выражение можно опустить – отсутствующее условное выражение будет интерпретироваться как `true`. Инструкция `for` имеет следующие ограничения:

- цикл может иметь только один индекс;
- индекс может иметь тип `int` или `float`;
- *выражение-изменения-индекса* должно иметь одну из следующих форм (предполагается, что индекс цикла имеет имя `i`):

`i++` , `i--` , `i += константное-выражение`, `i -= константное-выражение`

- *условное-выражение* должно выполнять сравнение индекса цикла с целочисленным константным выражением (см. раздел «Массивы»);
- в теле цикла не допускается выполнять присваивание индексу цикла.

Эти ограничения предусмотрены с целью дать компилятору возможность выполнять расширение встраиваемых инструкций `for`.

Инструкции `continue`, `break`, `discard`

Так же как в JavaScript или C, инструкции `continue` и `break` могут применяться только в теле цикла `for` и обычно используются в паре с инструкцией `if`:

- инструкция `continue` пропускает оставшуюся часть тела цикла, содержащего ее, наращивает/уменьшает индекс цикла и переходит в начало следующей итерации;

- инструкция `break` осуществляет выход из цикла, содержащего ее; после нее не выполняются никакие другие инструкции в цикле.

Ниже приводится пример использования инструкции `continue`:

```
for (int i = 0; i < 10; i++) {
    if (i == 8) {
        continue; // пропустить оставшуюся часть тела цикла
    }
    // эта строка не выполняется для i == 8
}
```

Ниже приводится пример использования инструкции `break`:

```
for (int i = 0; i < 10; i++) {
    if (i == 8) {
        break; // exits "for" loop
    }
    // эта строка не выполняется для i >= 8
}
// эта строка выполняется, если i == 8
```

Инструкцию `discard` можно использовать только внутри фрагментных шейдеров – она отбрасывает текущий фрагмент, отменяет операции с текущим фрагментом и выполняет переход к следующему фрагменту. Подробнее об инструкции `discard` рассказывается в разделе «Создание круглой точки» в главе 10, «Продвинутые приемы».

Функции

Функции в языке GLSL ES определяются точно так же, как в языке C. Например:

```
типВозвращаемогоЗначения имяФункции(тип0 arg0, тип1 arg1, ..., типn argn){
    некоторые вычисления
    return возвращаемоеЗначение;
}
```

Аргументы могут иметь любые типы, описываемые в этой главе. Допускается также объявлять функции, не имеющие аргументов (как, например, функция `main()`). Если функция не имеет возвращаемого значения, инструкцию `return` можно опустить. В этом случае `типВозвращаемогоЗначения` должен быть `void`. В качестве `возвращаемогоЗначения` допускается использовать структуры, с условием, что структура не имеет членов-массивов.

В следующем примере демонстрируется функция преобразования значения цвета RGBA в значение светимости (`luminance`):

```
float luma (vec4 color) {
    float r = color.r;
    float g = color.g;
    float b = color.b;
    return 0.2126 * r + 0.7162 * g + 0.0722 * b;
}
```

```
// Предыдущие четыре строки можно переписать иначе:  
// return 0.2126 * color.r + 0.7162 * color.g + 0.0722 * color.b;  
}
```

Вызов функций (таких, как в примере выше) выполняется точно так же, как в JavaScript или C, по имени, за которым следует список аргументов в круглых скобках:

```
attribute vec4 a_Color; // передаются (r, g, b, a)  
void main() {  
    ...  
    float brightness = luma(a_Color);  
    ...  
}
```

Обратите внимание, что несовпадение типов аргументов в вызове с типами параметров в объявлении функции приведет к ошибке. Например, следующий пример вызовет ошибку, потому что параметр имеет тип `float`, а вызывающий код пытается передать аргумент типа `int`:

```
float square(float value) {  
    return value * value;  
}  
  
void main() {  
    ...  
    float x2 = square(10); // Ошибка: 10 - это целое число. Должно быть 10.0.  
    ...  
}
```

Как следует из предыдущих примеров, функции GLSL ES действуют точно так же, как в JavaScript или C. Единственное исключение: функции не могут вызывать себя сами (то есть, рекурсивные вызовы недопустимы). Для технически подкованных читателей отметим, что это ограничение объясняется возможностью компилятора выполнять подстановку тела функции в точку ее вызова.

Объявления прототипов

Когда функция вызывается перед ее определением, необходимо объявить прототип функции. Прототип функции сообщает системе WebGL типы параметров и возвращаемого значения. Отметим, что этим GLSL ES отличается от JavaScript, где не требуется объявлять прототипы. Ниже приводится пример объявления прототипа для функции `luma()`, определение которой было представлено в предыдущем разделе:

```
float luma(vec4); // объявление прототипа  
  
main() {  
    ...  
    float brightness = luma(color); // luma() вызывается до того, как будет определена.  
    ...  
}
```

```

}

float luma (vec4 color) {
    return 0.2126 * color.r + 0.7162 * color.g + 0.0722 * color.b;
}

```

Квалификаторы параметров

В GLSL ES поддерживаются квалификаторы параметров, определяющие их роли внутри функции. С помощью квалификаторов можно указать, что параметр (1) является входным значением для функции, (2) является выходным значением для функции и (3) является одновременно и входным, и выходным значением для функции. В случаях (2) и (3) параметры используются подобно указателям в C. Все поддерживаемые квалификаторы перечислены в табл. 6.12.

Таблица 6.12. Квалификаторы параметров

Квалификатор	Роль	Описание
in	Входное значение для функции	Параметр передается по значению. Функция может читать и изменять его значение, но изменения будут недоступны вызывающему коду.
const in	Входное значение для функции	Параметр передается по значению. Функция может читать, но не может изменять его значение.
out	Выходное значение для функции	Параметр передается по ссылке. Может изменяться внутри функции и изменения будут доступны вызывающему коду.
inout	Входное/выходное значение для функции	Параметр передается по ссылке и его значение копируется внутри функции. Может изменяться внутри функции и изменения будут доступны вызывающему коду.
<нет: по умолчанию>	Входное значение для функции	Подобно квалификатору in.

Например, функция `luma()` могла бы вернуть результат вычислений через параметр, отмеченный квалификатором `out`, как показано ниже:

```

void luma2 (in vec3 color, out float brightness) {
    brightness = 0.2126 * color.r + 0.7162 * color.g + 0.0722 * color.b;
}

```

Так как в этом случае сама функция не имеет возвращаемого значения, ее тип изменился с `float` на `void`. Кроме того, квалификатор `in` перед первым параметром можно опустить, потому что он подразумевается по умолчанию.

Эту версию функции можно использовать следующим образом:

```

luma2(color, brightness); // результат хранится в "brightness"
// то же, что и brightness = luma(color)

```


Встроенные функции

Помимо функций, определяемых пользователем, GLSL ES поддерживает множество встроенных функций, реализующих часто используемые операции. В табл. 6.13 приводится краткая сводка по встроенным функциям в GLSL ES, а более подробные сведения о каждой из них вы найдете в приложении B.

Таблица 6.13. Встроенные функции в GLSL ES

Категория	Функции
Функции для работы с угловыми величинами	<code>radians</code> (преобразует градусы в радианы), <code>degrees</code> (преобразует радианы в градусы)
Тригонометрические функции	<code>sin</code> (синус), <code>cos</code> (косинус), <code>tan</code> (тангенс), <code>asin</code> (арксинус), <code>acos</code> (арккосинус) и <code>atan</code> (арктангенс)
Экспоненциальные функции	<code>pow</code> (x^y), <code>exp</code> (степень числе e), <code>log</code> (натуральный логарифм), <code>exp2</code> (2^x), <code>log2</code> (логарифм по основанию 2), <code>sqrt</code> (корень квадратный) и <code>inversesqrt</code> (обратный квадратный корень)
Общие функции	<code>abs</code> (абсолютное значение), <code>min</code> (минимальное значение), <code>max</code> (максимальное значение), <code>mod</code> (остаток от деления), <code>sign</code> (знак числа), <code>floor</code> (округление вниз), <code>ceil</code> (округление вверх), <code>clamp</code> (ограничивает значение заданным диапазоном), <code>mix</code> (линейная интерполяция), <code>step</code> (переходная функция), <code>smoothstep</code> (эрмитова интерполяция) и <code>fract</code> (дробная часть аргумента)
Геометрические функции	<code>length</code> (длина вектора), <code>distance</code> (расстояние между двумя точками), <code>dot</code> (внутреннее произведение), <code>cross</code> (внешнее произведение), <code>normalize</code> (вектор единичной длины), <code>reflect</code> (отражение вектора) и <code>faceforward</code> (преобразование нормали)
Матричные функции	<code>matrixCompMult</code> (покомпонентное умножение)
Векторные функции	<code>lessThan</code> (покомпонентное сравнение «<»), <code>lessThanEqual</code> (покомпонентное сравнение «<=»), <code>greaterThan</code> (покомпонентное сравнение «>»), <code>greaterThanEqual</code> (покомпонентное сравнение «>=»), <code>equal</code> (покомпонентное сравнение «==»), <code>notEqual</code> (покомпонентное сравнение «!=»), <code>any</code> (<code>true</code> , если хотя бы один компонент имеет значение <code>true</code>), <code>all</code> (<code>true</code> , если все компоненты имеют значение <code>true</code>) и <code>not</code> (покомпонентное сравнение логическое дополнение)
Функции для работы с текстурами	<code>texture2D</code> (поиск текселя в 2-мерной текстуре), <code>textureCube</code> (поиск текселя в кубической текстуре), <code>texture2DProj</code> (проективная версия <code>texture2D()</code>), <code>texture2DLod</code> (версия <code>texture2D()</code> с возможностью определять уровень детализации), <code>textureCubeLod</code> (версия <code>textureCube()</code> с возможностью определять уровень детализации) и <code>texture2DProjLod</code> (проективная версия <code>texture2DLod()</code>)

Глобальные и локальные переменные

Так же как JavaScript или C, GLSL ES поддерживает глобальные и локальные переменные. Глобальные переменные доступны из любого места в программе, а локальные – только внутри ограниченной области.

В GLSL ES, так же как в JavaScript или C, переменные, объявленные «за пределами» функции, являются глобальными, а объявленные «внутри» функции – локальными. Локальные переменные доступны только внутри функций, где они объявлены. Соответственно переменные-атрибуты, uniform-переменные и varying-переменные, описываемые в следующем разделе, должны объявляться как глобальные, так как они должны быть доступны за пределами функции.

Квалификаторы класса хранения

Как уже говорилось в предыдущих главах, в языке GLSL ES поддерживаются квалификаторы класса хранения для объявления переменных-атрибутов, uniform-переменных и varying-переменных (см. рис. 6.3). Кроме того, поддерживается квалификатор `const` для объявления переменных-констант.

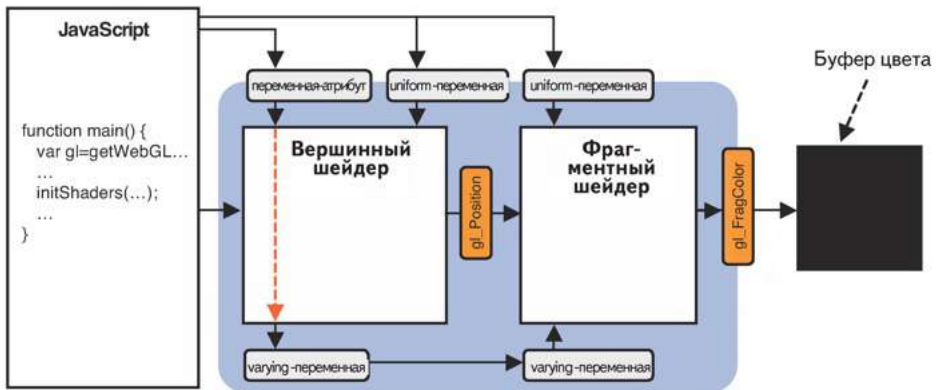


Рис. 6.3. Переменные-атрибуты, uniform-переменные и varying-переменные

Квалификатор `const`

В отличие от JavaScript, в GLSL ES поддерживается квалификатор `const`, позволяющий объявлять переменные-константы, то есть переменные, значение которых нельзя изменить.

Квалификатор `const` должен предшествовать объявлению типа переменной. Переменные с квалификатором `const` должны инициализироваться в момент объявления; в противном случае они окажутся бесполезными, потому что изменить их значение после объявления невозможно. Ниже приводится несколько примеров использования квалификатора `const`:

```
const int lightspeed = 299792458;           // скорость света (м/сек)
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0); // красный
const mat4 identity = mat4(1.0);           // единичная матрица
```

Попытка присвоить значение переменной, отмеченной квалификатором `const`, вызовет ошибку. Например:

```
const int lightspeed;
lightspeed = 299792458;
```

вследствие чего появится следующее сообщение:

```
failed to compile shader: ERROR: 0:11: 'lightspeed' : variables
with qualifier 'const' must be initialized
ERROR: 0:12: 'assign': l-value required (can't modify a const variable)
```

(Перевод: ошибка компиляции шейдера: ОШИБКА: 0:11: 'lightspeed' : переменные с квалификатором 'const' должны инициализироваться
ОШИБКА: 0:12: 'присваивание': необходимо l-значение (нельзя изменить переменную-константу))

Переменные-атрибуты

Как не раз говорилось в предыдущих главах, переменные-атрибуты доступны только в вершинных шейдерах. Они должны объявляться как глобальные переменные и используются для передачи в вершинный шейдер информации о каждой вершине. Обратите внимание на слова «о каждой вершине». Например, если имеется две вершины, (4.0, 3.0, 6.0) и (8.0, 3.0, 0.0), через переменную-атрибут должны быть переданы данные о каждой из них. Однако данные для других координат, таких как (6.0, 3.0, 3.0), соответствующих середине отрезка, соединяющего две вершины, нельзя передать через переменную. Если вам потребуется это, вы должны будете добавить еще одну вершину с этими координатами. Переменные-атрибуты можно использовать только для передачи данных следующих типов: float, vec2, vec3, vec4, mat2, mat3 и mat4. Например:

```
attribute vec4 a_Color;
attribute float a_PointSize;
```

Максимально возможное число переменных атрибутов зависит от конкретной реализации, но оно не может быть меньше 8. В табл. 6.14 указаны ограничения на число переменных каждого типа.

Таблица 6.14. Ограничения на число переменных-атрибутов, uniform-переменных и varying-переменных

Типы переменных		Встроенные константы, представляющие максимальное число	Минимальное число
переменные-атрибуты		const mediump int gl_MaxVertexAttribs	8
uniform-переменные	вершинный шейдер	const mediump int gl_MaxVertexUniformVectors	128
	фрагментный шейдер	const mediump int gl_MaxFragmentUniformVectors	16
varying-переменные		const mediump int gl_MaxVaryingVectors	8

uniform-переменные

uniform-переменные могут использоваться в шейдерах обоих видов – вершинных и фрагментных – и должны объявляться как глобальные переменные. uniform-переменные доступны только для чтения и могут объявляться с любыми типами, кроме массивов и структур. Если uniform-переменная одного и того же типа и с одним и тем же именем объявлена в обоих шейдерах, вершинном и фрагментном, она будет совместно использоваться ими. uniform-переменные предназначены для передачи «общих» (uniform) данных, поэтому в программах на JavaScript они должны использоваться только для передачи таких данных. Например, матрицы преобразований содержат данные, общие для всех вершин, поэтому их можно передавать в uniform-переменных:

```
uniform mat4 u_ViewMatrix;  
uniform vec3 u_LightPosition;
```

Максимально возможное число uniform-переменных зависит от реализации (см. табл. 6.14). Обратите внимание, что для вершинных и фрагментных шейдеров минимальное число таких переменных отличается.

varying-переменные

Последний квалификатор класса хранения – varying. varying-переменные также должны объявляться как глобальные и использоваться для передачи из вершинного шейдера в фрагментный, а это означает, что в обоих шейдерах должны объявляться одноименные переменные одного и того же типа. (См. объявление `v_Color` в листинге 6.1 и в листинге 6.2.) Ниже приводятся примеры объявления varying-переменных:

```
varying vec2 v_TexCoord;  
varying vec4 v_Color;
```

Подобно переменным-атрибутам, varying-переменные могут быть объявлены только со следующими типами: `float`, `vec2`, `vec3`, `vec4`, `mat2`, `mat3` и `mat4`. Как описывалось в главе 5, значение varying-переменной, присвоенное ей в вершинном шейдере, не передается во фрагментный шейдер непосредственно. Вместо этого между вершинным и фрагментным шейдерами выполняется процедура растеризации, осуществляющая интерполяцию значения и передачу интерполированного значения для каждого фрагмента в отдельности. Интерполяция как раз и является причиной ограниченного числа поддерживаемых типов данных, которые могут храниться в varying-переменных.

Максимально возможное число varying-переменных также зависит от реализации. Но оно не может быть меньше 8 (см. табл. 6.14).

Квалификаторы точности

Квалификаторы точности были введены в язык GLSL ES с целью увеличить скорость работы шейдеров и уменьшить потребление памяти. Как следует из назва-

ния, с помощью этого механизма можно ограничить точность (число битов) представления любого типа данных. Для обработки данных с более высокой точностью требуется больше памяти и времени, с меньшей точностью – меньше. Используя квалификаторы точности можно регулировать быстродействие и потребление памяти. Однако, квалификаторы точности могут поддерживаться не всеми реализациями, поэтому весьма разумно использовать их, как показано ниже:

```
#ifdef GL_ES
precision mediump float;
#endif
```

Так как спецификация WebGL разрабатывалась на основе спецификации OpenGL ES 2.0, которая регламентирует поддержку OpenGL на бытовых и встраиваемых устройствах, WebGL-программы могут выполняться на самом широком диапазоне устройств. Иногда скорость вычислений и потребление памяти можно улучшить за счет снижения точности представления данных. Что еще более важно, снижение точности позволяет также уменьшить потребление заряда аккумулятора и тем самым продлить период от зарядки до зарядки.

Однако, имейте в виду, что простое снижение точности может приводить к появлению неправильных результатов внутри WebGL, поэтому очень важно уметь находить баланс между эффективностью и правильностью вычислений.

Как показано в табл. 6.15, WebGL поддерживает три квалификатора точности: `highp` (high precision – наивысшая точность), `mediump` (medium precision – средняя точность) и `lowp` (lower precision – низшая точность).

Таблица 6.15. Квалификаторы точности

Квалификаторы точности	Описание	Диапазон по умолчанию и точность	
		float	int
<code>highp</code>	Высшая точность. Минимальная точность, требуемая в вершинных шейдерах.	$(-2^{62}, 2^{62})$ Точность: 2^{-16}	$(-2^{16}, 2^{16})$
<code>mediump</code>	Средняя точность. Минимальная точность, требуемая во фрагментных шейдерах. Выше, чем <code>lowp</code> и ниже, чем <code>highp</code> .	$(-2^{14}, 2^{14})$ Точность: 2^{-10}	$(-2^{10}, 2^{10})$
<code>lowp</code>	Низшая точность. Ниже, чем <code>mediump</code> , но достаточная для представления всех цветов.	$(-2, 2)$ Точность: 2^{-8}	$(-2^8, 2^8)$

Здесь необходимо сделать пару замечаний. Во-первых, в некоторых реализациях WebGL фрагментные шейдеры могут не поддерживать точность `highp`; как это проверить мы покажем далее в этом разделе. Во-вторых, фактический диапазон представляемых значений и точность зависят от реализации, проверить которые можно с помощью `gl.getShaderPrecisionFormat()`.

Ниже приводятся примеры объявления переменных с использованием квалификаторов точности:

```
mediump float size; // вещественное число средней точности
highp vec4 position; // вектор vec4 вещественных чисел высшей точности
lowp vec4 color; // вектор vec4 вещественных чисел низшей точности
```

Так как определение точности для каждой переменной может оказаться утомительной задачей, имеется возможность с помощью ключевого слова `precision` определить точность по умолчанию сразу для всего типа данных, что следует сделать в начале вершинного или фрагментного шейдера, используя следующий синтаксис:

```
precision квалификатор-точности имя-типа;
```

В результате точность, определяемая *квалификатором-точности*, будет установлена как точность по умолчанию для всех данных типа *имя-типа*. В этом случае, переменные, объявленные без квалификатора точности, получают точность, установленную по умолчанию. Например:

```
precision mediump float; // для всех вещественных чисел будет
                          // использоваться средняя точность
precision highp int; // для всех целых чисел будет
                     // использоваться наивысшая точность
```

В результате таких объявлений для всех типов данных, так или иначе связанных с типом `float`, таких как `vec2` и `mat3`, будет установлена средняя точность представления, а для всех типов данных, так или иначе связанных с типом `int`, будет установлена наивысшая точность представления. Например, так как вектор типа `vec4` содержит четыре значения типа `float`, для каждого значения в векторе будет установлена средняя точность представления.

Возможно вы обратили внимание, что в примерах фрагментных шейдеров, демонстрировавшихся в предыдущих главах, мы не применяли квалификаторы точности к типам данных, отличным от `float`. Это обусловлено тем, что для большинства типов данных устанавливается точность по умолчанию; однако, для типа `float` во фрагментных шейдерах точность по умолчанию не определена. Значения точности по умолчанию перечислены в табл. 6.16.

Таблица 6.16. Точность по умолчанию для разных типов

Шейдер	Тип данных	Точность по умолчанию
Вершинный	<code>int</code>	<code>highp</code>
	<code>float</code>	<code>highp</code>
	<code>sampler2D</code>	<code>lowp</code>
	<code>samplerCube</code>	<code>lowp</code>
Фрагментный	<code>int</code>	<code>medium</code>
	<code>float</code>	Нет
	<code>sampler2D</code>	<code>lowp</code>
	<code>samplerCube</code>	<code>lowp</code>

Тот факт, что для типа `float` не определена точность по умолчанию, требует от программиста проявлять особое внимание при использовании вещественных чисел в своих фрагментных шейдерах. Так, например, если попытаться использовать вещественную переменную без указания точности, это приведет к появлению следующей ошибки:

```
failed to compile shader: ERROR: 0:1 : No precision specified for (float).  
(Перевод: ошибка компиляции шейдера: ОШИБКА: 0:1 : Не определена точность  
для (float).)
```

Как уже отмечалось, поддержка точности `highp` для фрагментного шейдера зависит от конкретной реализации WebGL. Если она поддерживается, следовательно определен встроенный макрос `GL_FRAGMENT_PRECISION_HIGH` (см. следующий раздел).

Директивы препроцессора

Язык GLSL ES поддерживает директивы препроцессора, являющиеся командами для препроцессора, который обрабатывает исходный текст программы перед фактической компиляцией. Все директивы начинаются со знака решетки (`#`). Ниже приводится пример из программы `ColoredPoints.js`:

```
#ifdef GL_ES  
precision mediump float;  
#endif
```

В этих строках сначала проверяется, определен ли макрос `GL_ES`, и если определен, выполняются строки между директивами `#ifdef` и `#endif`. Директива `#ifdef` напоминает инструкцию `if` в JavaScript или C.

В GLSL ES доступны следующие три директивы препроцессора:

```
#if константное-выражение  
Если константное-выражение истинно, выполняется эта часть.  
#endif
```

```
#ifdef макрос  
Если макрос определен, выполняется эта часть.  
#endif
```

```
#ifndef макрос  
Если макрос не определен, выполняется эта часть.  
#endif
```

Определение макросов осуществляется с помощью директивы `#define`. В отличие от C, макросы в GLSL ES не могут иметь параметров:

```
#define имя-макроса строка
```

Удалить макрос можно с помощью директивы `#undef`:

```
#undef имя-макроса
```

Для определения альтернативных веток условной компиляции можно использовать директиву `#else`, по аналогии с инструкцией `if` в JavaScript или C. Например:

```
#define NUM 100
#if NUM == 100
Если NUM == 100, выполняется эта часть.
#else
Если NUM != 100, выполняется эта часть.
#endif
```

Макросам можно давать любые имена, кроме имен предопределенных макросов, которые перечислены в табл. 6.17.

Таблица 6.17. Предопределенные макросы

Макрос	Описание
GL_ES	Определен и имеет значение 1 в OpenGL ES 2.0
GL_FRAGMENT_PRECISION_HIGH	Точность highp поддерживается во фрагментных шейдерах

Макросы можно использовать с директивами препроцессора, как показано ниже:

```
#ifdef GL_ES
#ifdef GL_FRAGMENT_PRECISION_HIGH
precision highp float; // highp поддерживается. Назначить высшую точность
#else
precision mediump float; // highp не поддерживается. Назначить среднюю точность
#endif
#endif
```

Указать в шейдере, какая версия GLSL ES используется, можно с помощью директивы `#version`:

```
#version номер
```

Допустимыми номерами версий являются: 100 (для GLSL ES 1.00) и 101 (для GLSL ES 1.01). В отсутствие директивы `#version` будет предполагаться, что используется версия GLSL ES 1.00. Ниже приводится пример определения версии 1.01:

```
#version 101
```

Директива `#version` должна находиться в начале шейдера и ей могут предшествовать только пустые строки и строки комментариев.

В заключение

В этой главе рассказывалось об основных особенностях языка программирования шейдеров OpenGL ES Shading Language (GLSL ES).

Как вы имели возможность убедиться, язык шейдеров GLSL ES во многом похож на язык C, но предназначен для программирования компьютерной графики и из него убраны ненужные особенности, присущие языку C. В число специализированных особенностей входят: поддержка векторов и матриц, специальные имена компонентов для доступа к элементам векторов и матриц, а также операторы для работы с векторами и матрицами. Кроме того, GLSL ES поддерживает множество встроенных функций для выполнения операций, часто используемых при программировании компьютерной графики. Все это позволяет писать высокоэффективные шейдеры.

Теперь, когда вы получили более полное представление о языке GLSL ES, вернемся обратно в мир WebGL и исследуем еще более сложные примеры применения новых знаний.



ГЛАВА 7.

Вперед, в трехмерный мир

В предыдущих главах мы демонстрировали особенности работы системы WebGL, поддержку шейдеров, приемы преобразований, таких как трансляция и вращение, анимацию и наложение текстур на примере двумерной графики. Однако все приемы, с которыми мы познакомились, с успехом можно применять не только для рисования двумерных геометрических фигур, но и для отображения трехмерных объектов. В этой главе мы сделаем первый шаг в трехмерный мир и исследуем возможность перехода от двумерной к трехмерной графике. В частности, в этой главе мы исследуем:

- представление положения пользователя в трехмерном мире;
- управление восприятием глубины трехмерного пространства;
- отсечка;
- работа с объектами переднего и заднего плана;
- рисование трехмерного объекта (куба).

Все эти вопросы оказывают большое влияние на приемы рисования трехмерных сцен, владение которыми является залогом к созданию привлекательных трехмерных сцен. Как обычно, мы будем двигаться вперед небольшими шагами, что поможет вам быстро овладеть основами и подготовиться к решению более сложных задач, связанных с освещением и производительностью, которые рассматриваются в последующих главах.

Что хорошо для треугольников, хорошо и для кубов

До сих пор в большинстве наших исследований использовался скромный треугольник. Как уже говорилось выше, трехмерные объекты конструируются из двумерных фигур, в том числе и из треугольников. На рис. 7.1 изображен куб, составленный из 12 треугольников.

То есть, конструируя трехмерные объекты, можно применять те же самые приемы к треугольникам, составляющим эти объекты. Единственное и самое большое отличие от предыдущих примеров состоит в том, что теперь необходимо принимать во внимание третье измерение – **глубину** – в дополнение к координатам X и

У. Давайте для начала узнаем, как управлять направлением наблюдения (то есть, точкой, откуда пользователь наблюдает трехмерную сцену, и направлением, куда он смотрит), а затем – как управлять видимыми границами сцены, то есть, насколько глубоко эту сцену видит пользователь. В своих исследованиях мы будем опираться на простые треугольники, потому что так проще; однако, все, что истинно для треугольников, истинно и для трехмерных объектов.

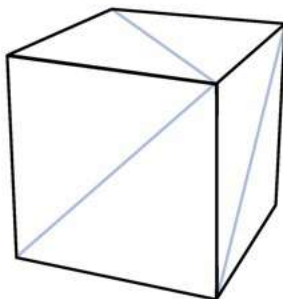


Рис. 7.1. Куб, составленный из 12 треугольников

Определение направления взгляда

Важнейшим фактором при работе с трехмерными объектами является необходимость учета третьего измерения – глубины. Это означает, что вам придется решать проблемы, которые отсутствовали при создании двухмерных фигур. Во-первых, как предполагает сама природа трехмерного мира, вы можете наблюдать объект из любой точки в пространстве; то есть, точка наблюдения может находиться где угодно. Описывая способ наблюдения объектов, необходимо учесть два важных момента:

- направление взгляда (куда направлен взгляд и какую часть сцены вы рассматриваете);
- видимый диапазон (границы поля зрения).

В этом первом разделе мы поближе познакомимся с понятием точки наблюдения и исследуем приемы, позволяющие помещать точку наблюдения в любое местоположение в трехмерном пространстве и рассматривать объекты с разных направлений. В следующем разделе мы перейдем ко второму пункту из списка выше и узнаем, как определить видимые границы сцены.

Как рассказывалось в главе 2, «Первые шаги в WebGL» (см. рис. 2.16), по умолчанию глаз пользователя (точка наблюдения) находится в начале координат $(0, 0, 0)$ и смотрит вдоль оси Z , в сторону отрицательных значений (за плоскость экрана). В этом разделе мы будем учиться перемещать точку наблюдения из местоположения по умолчанию в другие местоположения и рассматривать треугольник оттуда.

Давайте напишем программу `LookAtTriangles`, которая помещает точку наблюдения в координаты $(0.20, 0.25, 0.25)$ и отображает три треугольника, как они должны выглядеть при направлении взгляда в точку начала координат $(0, 0, 0)$. Использование трех треугольников поможет лучше понять, что такое глубина

трехмерной сцены. На рис. 7.2 изображен скриншот программы LookAtTriangles, а также цвет и направление координаты Z для каждого треугольника.

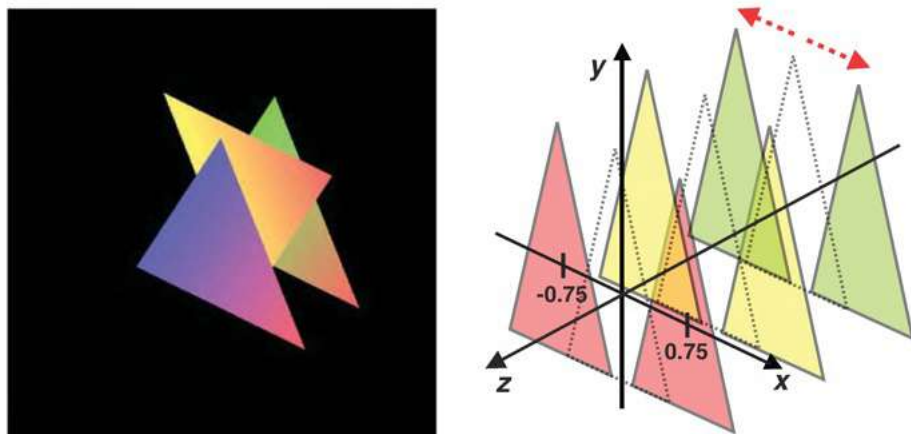


Рис. 7.2. Скриншот программы LookAtTriangles (слева), а также цвет и направление координаты Z для каждого треугольника (справа)

В программе используются мягкие, пастельные цвета, потому что они приятнее для восприятия.

Точка наблюдения, точка направления взгляда и направление вверх

Чтобы указать, откуда и куда направлен взгляд, необходимо определить координаты двух точек: точки наблюдения (откуда ведется наблюдение) и точки направления взгляда (часть сцены, куда направлен взгляд). Кроме того, в трехмерной компьютерной графике необходимо также указать, где находится верх сцены. По сути, чтобы определить направление наблюдения требуется указать всего три элемента информации (см. рис. 7.3).

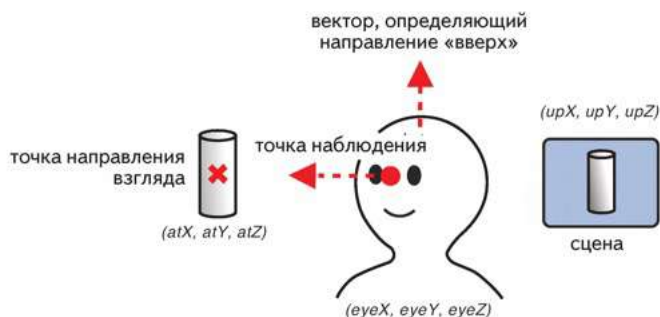


Рис. 7.3. Точка наблюдения, направление взгляда и направление вверх

Точка наблюдения: это начальная точка в трехмерном пространстве, откуда направлен взгляд. В следующих разделах координаты этой точки будут обозначаться как ($eyeX$, $eyeY$, $eyeZ$).

Точка направления взгляда: это точка, на которую направлен взгляд и которая определяет направления луча, исходящего из точки наблюдения. Как следует из названия, точка наблюдения – это всего лишь точка, а не вектор, поэтому, чтобы определить направление взгляда, необходима еще одна точка (точка направления взгляда). Точка направления взгляда – это точка, лежащая на луче, исходящем из точки наблюдения. Координаты этой точки будут обозначаться как (atX , atY , atZ).

Направление вверх: определяет направление вверх в сцене, рассматриваемой из точки наблюдения в указанном направлении. Если точка наблюдения и точка направления взгляда определены, мы свободно можем вращать сцену относительно луча, соединяющего точку наблюдения и точку направления взгляда. (На рис. 7.4 показано, что наклон головы наблюдателя вызывает смещение нижней и верхней границ сцены.) Чтобы определить вращение, необходимо определить направление вверх относительно луча зрения. Направление вверх задается тремя числами, представляющими направление. Координаты этого направления будут обозначаться как (upX , upY , upZ).



Рис. 7.4. Точка наблюдения, направление взгляда и направление вверх

В WebGL можно скомпоновать все эти координаты (точки наблюдения, точки направления взгляда и направления вверх) в общую матрицу и затем передавать эту матрицу в вершинный шейдер. Эта матрица называется матрицей преобразования вида, или просто **матрицей вида**, потому что она изменяет вид сцены. В библиотеке `cuon-matrix.js` имеется метод `Matrix4.setLookAt()`, вычисляющий матрицу вида на основе трех элементов информации: точки наблюдения, точки направления взгляда и направления вверх.

`Matrix4.setLookAt(eyeX, eyeY, eyeZ, atX, atY, atZ, upX, upY, upZ)`

Вычисляет матрицу вида, исходя из координат точки наблюдения ($eyeX$, $eyeY$, $eyeZ$), координат точки направления взгляда (atX , atY , atZ) и координат точки, определяющей направление вверх (upX , upY , upZ). Матрица вида возвращается в виде объекта `Matrix4`. Точка направления взгляда отображается в центр области рисования `<canvas>`.

Параметры	<code>eyeX, eyeY, eyeZ</code>	Координаты точки наблюдения.
	<code>atX, atY, atZ</code>	Координаты точки направления взгляда.

upX, upY, upZ

Координаты точки, определяющей направление вверх в сцене. Если направление вверх совпадает с положительным направлением оси Y, тогда координаты (upX, upY, upZ) этой точки определяются как (0, 1, 0).

Возвращаемое значение нет

Параметры `Matrix4.setLookAt()` в WebGL имеют следующие значения по умолчанию:

- точка наблюдения имеет координаты (0, 0, 0) (то есть, находится в начале системы координат);
- точка направления взгляда имеет отрицательную координату Z, то есть для нее отлично подходят координаты (0, 0, -1);¹
- направление вверх совпадает с положительным направлением оси Y, то есть ему отлично соответствуют координаты (0, 1, 0).

То есть, если, к примеру, направление вверх задать как (1, 0, 0), оно совпадет с положительным направлением оси X; в этом случае вы увидите сцену, повернутую на 90 градусов.

Матрицу вида, соответствующую параметрам по умолчанию, можно получить, как показано на рис. 7.5.

```
var initialViewMatrix = new Matrix4();
initialViewMatrix.setLookAt(0, 0, 0, 0, 0, -1, 0, 1, 0);
```

Рис. 7.5. Пример вызова `setLookAt()`

Теперь, после знакомства с методом `setLookAt()`, давайте посмотрим, как им пользоваться на практике.

Пример программы (LookAtTriangles.js)

В листинге 7.1 приводится исходный код программы `LookAtTriangles.js`, которая изменяет местоположение точки наблюдения и рисует три треугольника, как показано на рис. 7.2. Возможно, на бумаге это плохо видно: в порядке удаленности от наблюдателя треугольники имеют синий, желтый и зеленый цвет, соответственно, который плавно переходит в красный к правому нижнему углу.

Листинг 7.1. LookAtTriangles.js

```
1 // LookAtTriangles.js
2 // Вершинный шейдер
```

¹ Координата Z может иметь любое отрицательное значение. Значение -1 – это всего лишь пример, но вы можете выбрать любое другое отрицательное значение.

```

3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'attribute vec4 a_Color;\n' +
6   'uniform mat4 u_ViewMatrix;\n' +
7   'varying vec4 v_Color;\n' +
8   'void main() {\n' +
9   '   gl_Position = u_ViewMatrix * a_Position;\n' +
10  '   v_Color = a_Color;\n' +
11  '}\n';
12
13 // Фрагментный шейдер
14 var FSHADER_SOURCE =
15   ...
16   'varying vec4 v_Color;\n' +
17   'void main() {\n' +
18   '   gl_FragColor = v_Color;\n' +
19   '}\n';
20
21 function main() {
22   ...
23   // Определить координаты вершин и цвет (синий треугольник - первый)
24   var n = initVertexBuffers(gl);
25   ...
26   // Получить ссылку на переменную u_ViewMatrix
27   var u_ViewMatrix = gl.getUniformLocation(gl.program, 'u_ViewMatrix');
28   ...
29   // Точка наблюдения, точка направления взгляда и направление вверх
30   var viewMatrix = new Matrix4();
31   viewMatrix.setLookAt(0.20, 0.25, 0.25, 0, 0, 0, 0, 1, 0);
32
33   // Передать матрицу вида в переменную u_ViewMatrix
34   gl.uniformMatrix4fv(u_ViewMatrix, false, viewMatrix.elements);
35   ...
36   // Нарисовать треугольник
37   gl.drawArrays(gl.TRIANGLES, 0, n);
38 }
39
40 function initVertexBuffers(gl) {
41   var verticesColors = new Float32Array([
42     // координаты вершин и цвет
43     0.0, 0.5, -0.4, 0.4, 1.0, 0.4, // Дальний зеленый треугольник
44     -0.5, -0.5, -0.4, 0.4, 1.0, 0.4,
45     0.5, -0.5, -0.4, 1.0, 0.4, 0.4,
46
47     0.5, 0.4, -0.2, 1.0, 0.4, 0.4, // Желтый треугольник в середине
48     -0.5, 0.4, -0.2, 1.0, 1.0, 0.4,
49     0.0, -0.6, -0.2, 1.0, 1.0, 0.4,
50
51     0.0, 0.5, 0.0, 0.4, 0.4, 1.0, // Влижный синий треугольник
52     -0.5, -0.5, 0.0, 0.4, 0.4, 1.0,
53     0.5, -0.5, 0.0, 1.0, 0.4, 0.4
54   ]);
55   var n = 9;
56 }

```



```

88 // Создать буферный объект
89 var vertexColorbuffer = gl.createBuffer();
...
96 gl.bindBuffer(gl.ARRAY_BUFFER, vertexColorbuffer);
97 gl.bufferData(gl.ARRAY_BUFFER, verticesColors, gl.STATIC_DRAW);
...
121 return n;
122 }

```

Данный пример основан на программе `ColoredTriangle.js` из главы 5, «Цвет и текстура». Фрагментный шейдер, метод передачи информации о вершинах и некоторые другие части программы остались прежними. В новую версию внесено три главных изменения:

- матрица вида передается в вершинный шейдер (строка 6) и затем умножается на координаты вершин (строка 9);
- в функции `initVertexBuffers()` выполняется инициализация координат вершин и значений цвета трех треугольников (строки с 72 по 85), которая вызывается в строке 41, в функции `main()`;
- в строках 58 и 59 вычисляется матрица вида, которая затем в строке 62 передается через `uniform`-переменную `u_ViewMatrix` в вершинный шейдер; обратите внимание на координаты точки наблюдения (0.2, 0.25, 0.25), координаты точки направления взгляда (0, 0, 0) и координаты направления вверх (0, 1, 0).

Начнем с обзора второго изменения и функции `initVertexBuffers()` (строка 71). Разница между этой и оригинальной программой `ColoredTriangle.js` заключается в определении переменной `verticesColors` в строке 72 (которая в текущей версии является массивом координат вершин и значений цвета для трех треугольников) и появлении координат Z. Координаты и значения цвета сохраняются вместе, в буферном объекте `vertexColorBuffer` (строки 96 и 97), который создается в строке 89. Так как теперь мы имеем дело с тремя треугольниками (каждый из которых определяется тремя вершинами), мы должны передать методу `gl.drawArrays()` число 9 в третьем аргументе (строка 68).

Чтобы определить матрицу вида (то есть, местоположение точки наблюдения и точки направления взгляда [элемент 3]), ее требуется создать и передать в вершинный шейдер. Для этого в строке 58 создается объект `viewMatrix` типа `Matrix4`, и в строке 59 вызывается его метод `setLookAt()`, который вычисляет и сохраняет матрицу вида в `viewMatrix`. Передача матрицы вида через переменную `u_ViewMatrix` осуществляется в строке 62. Это – `uniform`-переменная, объявленная в вершинном шейдере:

```

57 // Точка наблюдения, точка направления взгляда и направление вверх
58 var viewMatrix = new Matrix4();
59 viewMatrix.setLookAt(0.20, 0.25, 0.25, 0, 0, 0, 0, 1, 0);
60
61 // Передать матрицу вида в переменную u_ViewMatrix
62 gl.uniformMatrix4fv(u_ViewMatrix, false, viewMatrix.elements);

```


Все эти изменения выполнены в коде на JavaScript. А теперь посмотрим, что происходит в вершинном шейдере:

```
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'attribute vec4 a_Color;\n' +
6   'uniform mat4 u_ViewMatrix;\n' +
7   'varying vec4 v_Color;\n' +
8   'void main() {\n' +
9     '  gl_Position = u_ViewMatrix * a_Position;\n' +
10    '  v_Color = a_Color;\n' +
11    '}\n';
```

Код вершинного шейдера начинается в строке 4. Единственные две строки, которыми этот вершинный шейдер отличается от оригинальной версии в `ColoredTriangle.js`, выделены жирным: в строке 6 определяется `uniform`-переменная `u_ViewMatrix`, и в строке 9 выполняется умножение матрицы вида на координаты вершины. Изменения выглядят очень простыми, поэтому возникает закономерный вопрос: как они влияют на местоположение точки наблюдения?

Сравнение *LookAtTriangles.js* с *RotatedTriangle_Matrix4.js*

Взглянув на вершинный шейдер в этом примере программы, можно заметить сходство с вершинным шейдером в программе `RotatedTriangle_Matrix4.js`, которая описывалась в главе 4, «Дополнительные преобразования и простая анимация». В том вершинном шейдере на основе объекта `Matrix4` создавалась матрица вращения, использовавшаяся затем для вращения треугольника. Давайте взглянем на тот шейдер еще раз:

```
1 // RotatedTriangle_Matrix4.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'uniform mat4 u_rotMatrix;\n' +
6   'void main() {\n' +
7     '  gl_Position = u_rotMatrix * a_Position;\n' +
8   '}\n';
```

Вершинный шейдер, представленный в этом разделе (`LookAtTriangles.js`), приводится ниже:

```
1 // LookAtTriangles.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'attribute vec4 a_Color;\n' +
6   'uniform mat4 u_ViewMatrix;\n' +
7   'varying vec4 v_Color;\n' +
```

```

8  'void main() {\n' +
9  '    gl_Position = u_ViewMatrix * a_Position;\n' +
10 '    v_Color = a_Color;\n' +
11 '}\n';

```

Как видите, были добавлены переменная-атрибут (`a_Color`) и `varying`-переменная для передачи значения цвета во фрагментный шейдер (`v_Color`), а также изменилось имя `uniform`-переменной с `u_rotMatrix` на `u_ViewMatrix`. Но, несмотря на эти изменения, порядок вычисления значения для переменной `gl_Position` остался прежним: матрица типа `mat4` умножается на вектор `a_Position`. (сравните строку 7 в `RotatedTriangle_Matrix4.js` со строкой 9 в `LookAtTriangles.js`.)

Из этого следует, что операция получения вида сцены «наблюдаемой с указанной точки, когда взгляд направлен в заданную точку» фактически эквивалентна операциям преобразований, таким как трансляция или вращение.

Давайте попробуем разобраться с этим на наглядном примере. Допустим, что мы сначала смотрим на треугольник из точки с координатами $(0, 0, 0)$ вдоль оси Z , в направлении отрицательных значений, а затем перемещаемся в точку с координатами $(0, 0, 1)$ (слева на рис. 7.6). В этом случае расстояние между точкой наблюдения и треугольником увеличивается на 1.0 единиц по оси Z . Того же эффекта можно добиться, если остаться в той же точке наблюдения и сместить треугольник на 1.0 единицу (справа на рис. 7.6).

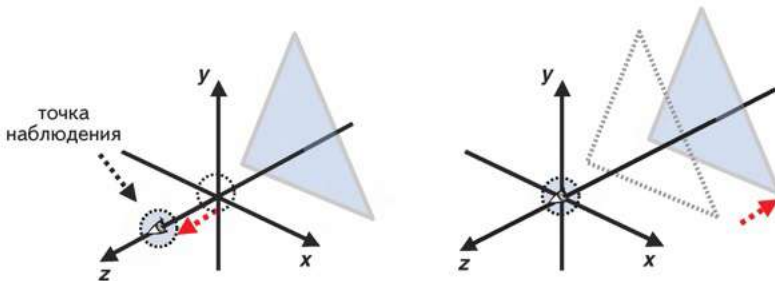


Рис. 7.6. Смещение точки наблюдения равноценно смещению наблюдаемого объекта

Именно это и происходит в нашей программе. Метод `setLookAt()` объекта `Matrix4` просто вычисляет матрицу, описывающую это преобразование, опираясь на информацию о координатах точки наблюдения, координатах точки, в которую направлен взгляд, и координатах, описывающих направление вверх. Соответственно, умножая матрицу на векторы координат вершин объектов сцены, мы получаем тот же эффект, что и смещение точки наблюдения. По сути, вместо перемещения точки наблюдения в одном направлении, мы перемещаем объекты сцены (то есть, сам мир) в противоположном направлении. Тот же подход можно использовать для реализации вращения точки наблюдения.

Так как перемещение точки наблюдения фактически описывается такими преобразованиями, как вращение или трансляция, их можно представить в виде ма-

трицы преобразования. Давайте посмотрим, как вычисляется матрица, когда необходимо повернуть треугольник и сместить точку наблюдения.

Взгляд на повернутый треугольник с указанной позиции

Программа `RotatedTriangle_Matrix4` в главе 4 отображает треугольник, повернутый относительно оси Z. В этом разделе демонстрируется измененная ее версия, программа `LookAtTriangles`, отображающая три треугольника, также повернутые относительно оси Z и видимые с указанной точки наблюдения. В данном примере потребовалось использовать две матрицы: матрицу вращения и матрицу вида, определяющую параметры позиции, с которой наблюдается сцена. Первая проблема, которую мы рассмотрим – порядок, в каком должны умножаться эти матрицы.

Как мы уже знаем, умножение матрицы на координаты вершины обеспечивает определяемое матрицей преобразование координат. То есть умножение матрицы поворота на координаты вершины вызывает поворот.

Умножение матрицы вида на координаты вершины обеспечивает преобразование, смещающее вершину в позицию, соответствующую положению наблюдателя. В данном примере требуется показать, как будут выглядеть повернутые треугольники с заданной точки наблюдения, поэтому мы сначала должны повернуть треугольники, а затем преобразовать сцену в соответствии с положением точки наблюдения. Иными словами, мы должны повернуть три вершины, составляющие треугольник, и затем преобразовать координаты вершин, полученные после вращения треугольника, чтобы показать, как они будут выглядеть с заданной позиции. Добиться этого можно, выполнив умножение матриц в порядке, описанном в предыдущем предложении. Рассмотрим соответствующие формулы.

Как описывалось прежде, если требуется повернуть фигуру, следует умножить матрицу вращения на координаты вершин фигуры:

$$\begin{aligned} &\langle \text{координаты вершины после поворота} \rangle = \\ &\quad \langle \text{матрица вращения} \rangle \times \langle \text{исходные координаты вершины} \rangle \end{aligned}$$

Умножая матрицу вида на координаты вершины, полученные после поворота, как описывает предыдущая формула, получаем координаты вершины после поворота, при наблюдении с заданной точки.

$$\begin{aligned} &\langle \text{координаты вершины после поворота, при наблюдении с заданной точки} \rangle = \\ &\quad \langle \text{матрица вида} \rangle \times \langle \text{координаты вершины после поворота} \rangle \end{aligned}$$

Подставив первое выражение во второе, получаем:

$$\begin{aligned} &\langle \text{координаты вершины после поворота, при наблюдении с заданной точки} \rangle = \\ &\quad \langle \text{матрица вида} \rangle \times \langle \text{матрица вращения} \rangle \\ &\quad \times \langle \text{координаты вершины после поворота} \rangle \end{aligned}$$

В данном выражении используется матрица вращения, но с тем же успехом можно использовать матрицу трансляции (переноса), матрицу масштабирования

или их комбинации. Такие матрицы в общем случае называют **матрицами модели** (model matrix). Используем этот термин и перепишем последнее выражение, как показано в формуле 7.1.

Формула 7.1.

$$\langle \text{матрица вида} \rangle \times \langle \text{матрица модели} \rangle \times \langle \text{координаты вершины} \rangle$$

Теперь нужно реализовать эту формулу, а так как это довольно простое выражение, его можно реализовать непосредственно в вершинном шейдере. На рис. 7.7 приводится скриншот программы `LookAtRotatedTriangles`, реализующий преобразования. Обратите внимание, что на этом рисунке белым пунктиром показано положение треугольника перед поворотом, чтобы вам проще было заметить поворот.

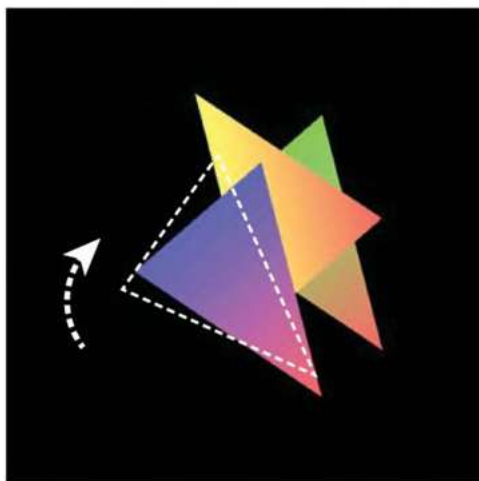


Рис. 7.7. `LookAtRotatedTriangles`

Пример программы (`LookAtRotatedTriangles.js`)

Программа `LookAtRotatedTriangles.js` – это немного измененная версия программы `LookAtTriangles.js`. Нам потребовалось добавить `uniform`-переменную `u_ModelMatrix` для передачи матрицы модели в шейдер и код в функцию `main()` на JavaScript, реализующий запись матрицы в переменную. Соответствующий код приводится в листинге 7.2.

Листинг 7.2. `LookAtRotatedTriangles.js`

```
1 // LookAtRotatedTriangles.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'attribute vec4 a_Color;\n' +
6   'uniform mat4 u_ViewMatrix;\n' +
```



```

7  'uniform mat4 u_ModelMatrix;\n' +
8  'varying vec4 v_Color;\n' +
9  'void main() {\n' +
10 '    gl_Position = u_ViewMatrix * u_ModelMatrix * a_Position;\n' +
11 '    v_Color = a_Color;\n' +
12 '}\n';
...
24 function main() {
    ...
51 // Получить ссылки на u_ViewMatrix и u_ModelMatrix
52 var u_ViewMatrix = gl.getUniformLocation(gl.program, 'u_ViewMatrix');
53 var u_ModelMatrix = gl.getUniformLocation(gl.program, 'u_ModelMatrix');
    ...
59 // Определить точку и линию наблюдения
60 var viewMatrix = new Matrix4();
61 viewMatrix.setLookAt(0.20, 0.25, 0.25, 0, 0, 0, 0, 1, 0);
62
63 // Вычислить матрицу вращения
64 var modelMatrix = new Matrix4();
65 modelMatrix.setRotate(-10, 0, 0, 1); // Поворот относительно оси Z
66
67 // Передать обе матрицы через отдельные uniform-переменные
68 gl.uniformMatrix4fv(u_ViewMatrix, false, viewMatrix.elements);
69 gl.uniformMatrix4fv(u_ModelMatrix, false, modelMatrix.elements);

```

Сначала рассмотрим вершинный шейдер. В строке 10 можно видеть простую реализацию формулы 7.1, использующую `u_ModelMatrix`, объявленную в строке 7, которая принимает данные от программы на JavaScript:

```

10 '    gl_Position = u_ViewMatrix * u_ModelMatrix * a_Position;\n' +

```

В JavaScript-функции `main()` уже имеется код, вычисляющий матрицу вида, поэтому нам нужно всего лишь добавить код для вычисления матрицы вращения, описывающей поворот относительно оси Z на -10 градусов. В строке 53 программа получает ссылку на переменную `u_ModelMatrix` и в строке 64 создает матрицу `modelMatrix` для хранения матрицы вращения. Далее, в строке 65, производится вычисление этой матрицы вызовом `Matrix4.setRotate()`, которая затем в строке 69 передается в переменную `u_ModelMatrix`.

Если запустить эту программу, она нарисует треугольники, как показано на рис. 7.6, иллюстрируя, что умножение матриц на координаты вершин (`a_Position`) дает желаемый эффект. То есть, сначала осуществляется поворот вершин в соответствии с матрицей `u_ModelMatrix`, а затем, путем применения матрицы `u_ViewMatrix`, сцена преобразуется к виду, наблюдаемому с указанной позиции.

Эксперименты с примером программы

В приложении `LookAtRotatedTriangles.js` реализована формула 7.1. Однако, из-за того, что умножение матриц вида и модели выполняется в вершинном шейдере для каждой вершины, данная реализация оказывается не самой эффективной, что особенно заметно при большом количестве вершин. Произведение матриц из

формулы 7.1 идентично для всех вершин, поэтому его можно вычислить заранее один раз и передавать в вершинный шейдер уже готовый результат. Матрица, полученная в результате умножения матрицы вида на матрицу модели, называется **матрицей модели вида** (model view matrix). То есть,

$$\langle \text{матрица модели вида} \rangle = \langle \text{матрица вида} \rangle \times \langle \text{матрица модели} \rangle$$

С учетом этого формулу 7.1 можно переписать, как показано в формуле 7.2.

Формула 7.2

$$\langle \text{матрица модели вида} \rangle \times \langle \text{координаты вершины} \rangle$$

В листинге 7.3 представлена версия программы LookAtRotatedTriangles_mvMatrix, реализующая формулу 7.2.

Листинг 7.3. LookAtRotatedTriangles_mvMatrix.js

```
1 // LookAtRotatedTriangles_mvMatrix.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4 ...
5
6 'uniform mat4 u_ModelViewMatrix;\n' +
7 'varying vec4 v_Color;\n' +
8 'void main() {\n' +
9   '  gl_Position = u_ModelViewMatrix * a_Position;\n' +
10  '  v_Color = a_Color;\n' +
11  '}\n';
12 ...
13 function main() {
14 ...
15
16 // Получить ссылки на u_ModelViewMatrix и u_ModelMatrix
17 var u_ModelViewMatrix = gl.getUniformLocation(gl.program, 'u_ModelViewMatrix');
18 ...
19
20 viewMatrix.setLookAt(0.20, 0.25, 0.25, 0, 0, 0, 0, 1, 0);
21 ...
22
23 modelMatrix.setRotate(-10, 0, 0, 1); // Вычислить матрицу вращения
24
25 // Умножить матрицы
26 var modelViewMatrix = viewMatrix.multiply(modelMatrix);
27
28 // Передать матрицу модели вида в u_ModelViewMatrix
29 gl.uniformMatrix4fv(u_ModelViewMatrix, false, modelViewMatrix.elements);
```

Имя uniform-переменной в вершинном шейдере, участвующей в вычислениях в строке 9, изменилось на u_ModelViewMatrix. Однако сам алгоритм работы вершинного шейдера остался прежним, как в оригинальной версии LookAtTriangles.js.

Вычисление матриц viewMatrix и modelMatrix в программе на JavaScript (строки с 59 по 63) выполняется с применением тех же методов, что и в программе LookAtRotatedTriangles.js, но затем производится умножение матриц с сохранением результата в modelViewMatrix (строка 66). Умножение выполняется вызовом метода multiply() объекта Matrix4. Он умножает матрицу

`viewMatrix` справа на матрицу, определяемую аргументом (`modelMatrix`). То есть в данной программе выполняется умножение `modelViewMatrix = viewMatrix * modelMatrix`, только в языке JavaScript, в отличие от GLSL ES, для умножения матриц необходимо использовать метод вместо простого оператора `*`.

После получения матрицы `modelViewMatrix` остается лишь передать ее в переменную `u_ModelViewMatrix` (строка 69). Запустив программу, можно наблюдать тот же результат, что показан на рис. 7.6.

И последнее замечание к этому примеру: каждая матрица в этой программе вычисляется по отдельности (строки 59, 63 и 66), чтобы вам проще было понять порядок вычислений. Однако эти вычисления можно оптимизировать, реализовав их в одной строке:

```
var modelViewMatrix = new Matrix4();
modelViewMatrix.setLookAt(0.20, 0.25, 0.25, 0, 0, 0, 0, 0, 1, 0).rotate(-10, 0, 0, 1);
// Передать матрицу модели вида в uniform-переменную
gl.uniformMatrix4fv(u_ModelViewMatrix, false, modelViewMatrix.elements);
```

Изменение точки наблюдения с клавиатуры

Давайте изменим программу `LookAtTriangles` так, чтобы имелась возможность изменять положение точки наблюдения с помощью клавиш со стрелками. В новой программе `LookAtTrianglesWithKeys` клавиша со стрелкой вправо увеличивает координату X на 0.01; клавиша со стрелкой влево – уменьшает на 0.01. На рис. 7.8 показан скриншот этой программы. Если нажать и удерживать клавишу со стрелкой влево, сцена изменится, как показано на рис. 7.8 справа.

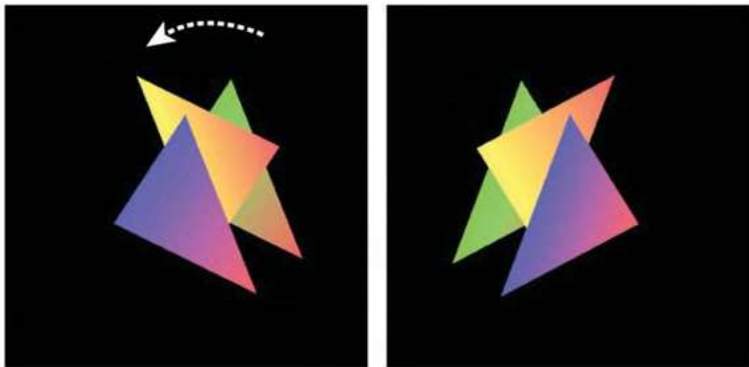


Рис. 7.8. `LookAtTrianglesWithKeys`

Пример программы (`LookAtTrianglesWithKeys.js`)

В листинге 7.4 приводится исходный код примера программы. Вершинный и фрагментный шейдеры остались прежними, как в программе `LookAtTriangles.js`. Основные вычисления, выполняемые в JavaScript-функции `main()`, так же не изменились. Главное отличие заключается в том, что в эту версию программы

был добавлен обработчик события, вызываемый при нажатии на клавишу, и код рисования треугольников был перемещен в функцию `draw()`.

Листинг 7.4. LookAtTrianglesWithKeys.js

```

1 // LookAtTrianglesWithKeys.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'attribute vec4 a_Color;\n' +
6   'uniform mat4 u_ViewMatrix;\n' +
7   'varying vec4 v_Color;\n' +
8   'void main() {\n' +
9     '  gl_Position = u_ViewMatrix * a_Position;\n' +
10    '  v_Color = a_Color;\n' +
11    '}\n';
12
13 ...
23 function main() {
14   ...
15   // Получить ссылку на переменную u_ViewMatrix
16   var u_ViewMatrix = gl.getUniformLocation(gl.program, 'u_ViewMatrix');
17   ...
18   // Создать объект Matrix4 для матрицы вида
19   var viewMatrix = new Matrix4();
20   // Зарегистрировать обработчик событий нажатий клавиш
21   document.onkeydown = function(ev){ keydown(ev, gl, n, u_ViewMatrix,
22                                             viewMatrix); };
23
24   draw(gl, n, u_ViewMatrix, viewMatrix); // Нарисовать треугольник
25 }
26
27 ...
117 var g_eyeX = 0.20, g_eyeY = 0.25, g_eyeZ = 0.25; // Точка наблюдения
118 function keydown(ev, gl, n, u_ViewMatrix, viewMatrix) {
119   if(ev.keyCode == 39) { // Нажата клавиша со стрелкой вправо?
120     g_eyeX += 0.01;
121   } else
122   if (ev.keyCode == 37) { // Нажата клавиша со стрелкой влево?
123     g_eyeX -= 0.01;
124   } else { return; } // Предотвратить ненужное перерисовывание
125   draw(gl, n, u_ViewMatrix, viewMatrix);
126 }
127
128 function draw(gl, n, u_ViewMatrix, viewMatrix) {
129   // Установить параметры позиции для точки наблюдения
130   viewMatrix.setLookAt(g_eyeX, g_eyeY, g_eyeZ, 0, 0, 0, 0, 1, 0);
131
132   // Передать матрицу вида в переменную u_ViewMatrix
133   gl.uniformMatrix4fv(u_ViewMatrix, false, viewMatrix.elements);
134
135   gl.clear(gl.COLOR_BUFFER_BIT); // Очистить <canvas>
136
137   gl.drawArrays(gl.TRIANGLES, 0, n); // Нарисовать треугольник
138 }

```


Для изменения позиции точки наблюдения мы использовали обработчик событий нажатий на клавиши со стрелками вправо и влево. Прежде чем приступить к исследованию обработчика событий, рассмотрим функцию `draw()`, которая вызывается из этого обработчика.

Функция `draw()` выполняет достаточно простую последовательность операций. В строке 130 она вычисляет матрицу вида, используя глобальные переменные `g_eyeX`, `g_eyeY` и `g_eyeZ`, которые объявлены в строке 117 и инициализированы значениями 0.2, 0.25 и 0.25, соответственно. Затем, в строке 133, матрица передается в вершинный шейдер через `uniform`-переменную `u_ViewMatrix`. Ссылка на переменную `u_ViewMatrix` приобретается в функции `main()`, в строке 51, а передаваемый в ней объект типа `Matrix4 (viewMatrix)` создается в строке 58. Эти две операции выполняются заранее, потому что было бы слишком расточительно выполнять их в каждой операции рисования. После этого в строке 135 производится очистка области рисования `<canvas>` и в строке 137 выполняется рисование треугольников.

Переменные `g_eyeX`, `g_eyeY` и `g_eyeZ` определяют местоположение точки наблюдения и для них вычисляются новые значения при каждом нажатии на обрабатываемые клавиши. Чтобы обеспечить вызов обработчика по событиям нажатия на клавиши, его необходимо зарегистрировать в свойстве `onkeydown` объекта `document`. Внутри обработчика событий осуществляется вызов функции `draw()`, которой требуется передать все необходимые аргументы. Именно поэтому обработчик регистрируется как анонимная функция:

```
59 // Зарегистрировать обработчик событий нажатий клавиш
60 document.onkeydown = function(ev){ keydown(ev, gl, n, u_ViewMatrix,
                                     viewMatrix); };
```

Она вызывает собственно обработчик `keydown()`. Теперь рассмотрим реализацию функции `keydown()`.

```
118 function keydown(ev, gl, n, u_ViewMatrix, viewMatrix) {
119   if(ev.keyCode == 39) { // Нажата клавиша со стрелкой вправо?
120     g_eyeX += 0.01;
121   } else
122   if (ev.keyCode == 37) { // Нажата клавиша со стрелкой влево?
123     g_eyeX -= 0.01;
124   } else { return; }      // Предотвратить ненужное перерисовывание
125   draw(gl, n, u_ViewMatrix, viewMatrix);
126 }
```

Функция `keydown()` также проста и понятна. Когда пользователь нажимает клавишу, происходит вызов функции `keydown()`, которой в первом параметре `ev` передается информация о событии. Нам остается только проверить значение свойства `ev.keyCode` и выяснить, какая клавиша была нажата, изменить при необходимости значение переменной `g_eyeX` и нарисовать треугольники. Если нажата клавиша со стрелкой вправо, значение переменной `g_eyeX` увеличивается на 0.01, если нажата клавиша со стрелкой влево, значение переменной `g_eyeX` уменьшается на 0.01.

Если запустить эту программу, можно заметить, что после каждого нажатия на клавиши со стрелками треугольники смещаются.

Недостающие части

Экспериментируя с программой, можно заметить, что при достаточно большом смещении влево или вправо часть треугольника исчезает (см. рис. 7.9).

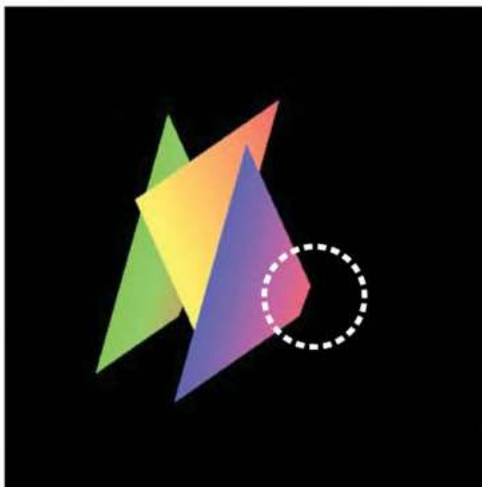


Рис. 7.9. Часть треугольника исчезает

Это объясняется тем, что мы не указали **видимый диапазон** (границы поля зрения). Как упоминалось в первом разделе этой главы, WebGL не отображает объекты, выходящие за видимый диапазон. На рис. 7.9 как раз показана ситуация, когда часть треугольника вышла за этот диапазон.

Определение видимого объема в форме прямоугольного параллелепипеда

Система WebGL позволяет размещать трехмерные объекты в любой точке трехмерного пространства, но отображает только те, которые оказываются внутри границ видимого диапазона. Такое ограничение обусловлено, в первую очередь, необходимостью обеспечения высокой производительности; нет смысла рисовать трехмерные объекты, если они невидимы для наблюдателя. Кроме того, такое решение в определенной степени имитирует ограниченность поля зрения человека (см. рис. 7.10); мы видим объекты в определенных границах, относительно линии зрения, попадающие в сектор, который имеет величину примерно 200 градусов по горизонтали. WebGL также определяет видимые границы и не отображает объекты, выходящие за них.

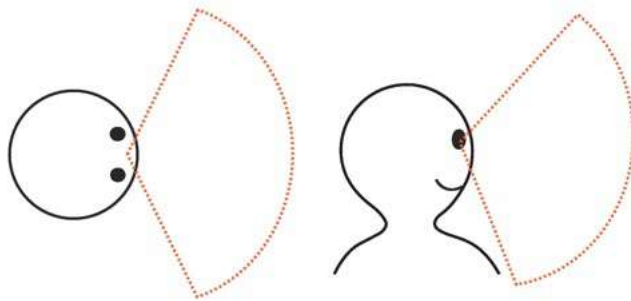


Рис. 7.10. Поле зрения человека

Помимо верхней/нижней и левой/правой границ, WebGL ограничивает также видимую глубину, имитируя ограниченную дальность нашего зрения. Все эти границы в совокупности определяют **видимый объем**. На рис. 7.9 часть треугольника исчезла, потому что глубина видимого объема оказалась недостаточной.

Определение видимого объема

Существует два способа определить видимый объем:

- в виде прямоугольного параллелепипеда (**ортогональная проекция**);
- в виде четырехгранной пирамиды (**перспективная проекция**).

Перспективная проекция позволяет лучше почувствовать глубину и часто обеспечивает более реалистичное изображение, так как полнее соответствует тому, как мы видим мир в действительности. Эту проекцию следует использовать, чтобы показать трехмерную сцену в перспективе, например, персонажи или поле битвы в трехмерной игре. Ортогональная проекция упрощает сравнение двух объектов, таких как две части модели молекулы, потому что отсутствует влияние такого фактора, как разность расстояний от точки наблюдения. Эту проекцию следует использовать, чтобы показать трехмерные объекты в ортогональной проекции, например, детали на техническом рисунке.

Сначала рассмотрим особенности видимого объема в ортогональной проекции.

На рис. 7.11 схематически изображен видимый объем в ортогональной проекции. Этот видимый объем отстоит на некотором удалении от точки наблюдения вдоль линии взгляда и занимает область пространства, ограниченную двумя плоскостями: **ближняя плоскость отсечения** и **дальняя плоскость отсечения**. Ближняя плоскость отсечения определяется границами: (*right, top, -near*), (*-left, top, -near*), (*-left, -bottom, -near*) и (*right, -bottom, -near*). Дальняя плоскость отсечения определяется границами: (*right, top, -far*), (*-left, top, -far*), (*-left, -bottom, -far*) и (*right, -bottom, -far*).

Видимая область сцены начинается от ближней плоскости отсечения и простирается вдоль линии взгляда до дальней плоскости отсечения. Именно она и рисуется в элементе `<canvas>`. Если соотношение сторон ближней плоскости отсечения будет отличаться от соотношения сторон области рисования элемента

<canvas>, WebGL произведет масштабирование и формы геометрических фигур в сцене будут искажены. (Эту особенность мы рассмотрим в конце данного раздела.) Пространство от ближней плоскости отсечения до дальней определяет видимый объем. Отображаться будут только объекты, находящиеся в этом объеме. Если объект какой-то своей частью выйдет за границы объема, WebGL отобразит только часть объекта, находящуюся внутри объема.

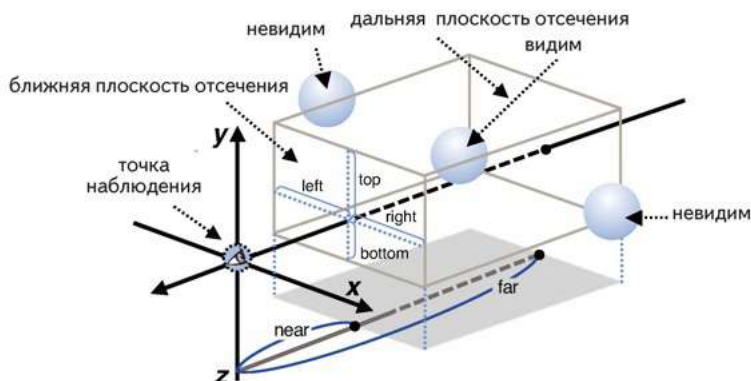


Рис. 7.11. Видимый объем в ортогональной проекции

Определение границ видимого объема в форме параллелепипеда

Для определения границ ортогонального видимого объема используется метод `setOrtho()`, поддерживаемый объектами типа `Matrix4`, который определяется в библиотеке `cuon-matrix.js`.

`Matrix4.setOrtho(left, right, bottom, top, near, far)`

Вычисляет матрицу (матрицу ортогональной проекции), которая определяет видимый объем, исходя из своих аргументов, и сохраняет ее в объекте `Matrix4`. При этом значение `left` не должно быть равно значению `right`, значение `bottom` не должно быть равно значению `top` и значение `near` не должно быть равно значению `far`.

Параметры	<code>left, right</code>	Определяют расстояние до левого и правого краев ближней плоскости отсечения.
	<code>bottom, top</code>	Определяют расстояние до нижнего и верхнего краев ближней плоскости отсечения.
	<code>near, far</code>	Определяют расстояние до ближней и дальней плоскости отсечения от точки наблюдения вдоль линии взгляда.

Возвращаемое значение нет

Здесь снова используется матрица, которая на этот раз называется **матрицей ортогональной проекции** (orthographic projection matrix). В примере `OrthoView` мы будем использовать матрицу проекции именно этого типа, чтобы определить видимый объем и затем нарисовать треугольники – как в программе `LookAtRotatedTriangles` – чтобы посмотреть, какое влияние оказывает определение границ видимого объема. В примере `LookAtRotatedTriangles` мы помещали точку наблюдения в местоположение, отличное от начала координат. Но в этом примере точка наблюдения будет находиться в позиции с координатами $(0, 0, 0)$ и взгляд будет направлен вдоль оси Z , в сторону отрицательных значений, что поможет нам упростить исследование влияния видимого объема. Видимый объем определяется в этом примере, как показано на рис. 7.12, где $near = 0.0$, $far = 0.5$, $left = -1.0$, $right = 1.0$, $bottom = -1.0$ и $top = 1.0$, потому что треугольники расположены в области пространства между координатами 0.0 и -0.4 по оси Z (см. рис. 7.2).

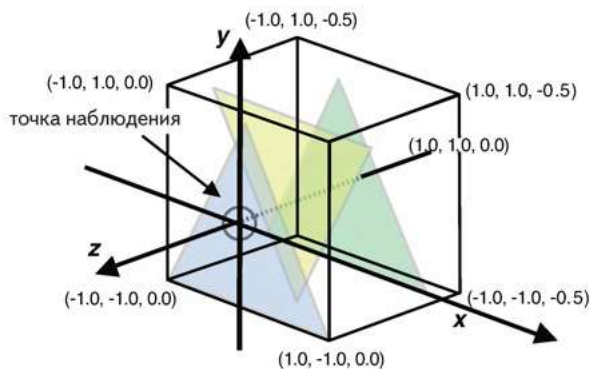


Рис. 7.12. Ортогональный видимый объем, используемый в `OrthoView`

В этот пример мы также добавили в обработчик событий поддержку изменения значений $near$ и far , чтобы дать возможность проверить эффект изменения размеров видимого объема. Ниже перечислены активные клавиши и их влияние.

Клавиша	Действие
Стрелка вправо	Увеличивает $near$ на 0.01
Стрелка влево	Уменьшает $near$ на 0.01
Стрелка вверх	Увеличивает far на 0.01
Стрелка вниз	Уменьшает far на 0.01

Чтобы вы могли видеть текущие значения $near$ и far , они выводятся под элементом `<canvas>`, как показано на рис. 7.13.

А теперь перейдем к примеру программы.



near: 0, far: 0.5

Рис. 7.13. OrthoView

Пример программы (OrthoView.html)

Так как эта программа отображает значения *near* и *far* на веб-странице, а не в области рисования `<canvas>`, необходимо внести некоторые изменения в файл HTML, как показано в листинге 7.5 (OrthoView.html).

Листинг 7.5. OrthoView.html

```
1 <!DOCTYPE html>
2 <html>
3   <head lang="ja">
4     <meta charset="utf-8" />
5     <title>Set Box-shaped Viewing Volume</title>
6   </head>
7
8   <body onload="main()">
9     <canvas id="webgl" width="400" height="400">
10       Please use a browser that supports <canvas>
11     </canvas>
12     <p id="nearFar"> The near and far values are displayed here. </p>
13
14     <script src="../lib/webgl-utils.js"></script>
15     ...
18     <script src="OrthoView.js"></script>
19   </body>
20 </html>
```

Как видите, в этом примере добавилась строка 12. В этой строке присутствует текст «The near and far values are displayed here» (Здесь отображаются значения

near и far), который будет замещаться из JavaScript текущими значениями *near* и *far*.

Пример программы (OrthoView.js)

В листинге 7.6 приводится исходный код программы OrthoView.js. Эта программа практически полностью совпадает с программой LookAtTrianglesWithKeys.js, добавилось лишь управление местоположением точки наблюдения с клавиатуры.

Листинг 7.6. OrthoView.js

```

1 // OrthoView.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'attribute vec4 a_Color;\n' +
6   'uniform mat4 u_ProjMatrix;\n' +
7   'varying vec4 v_Color;\n' +
8   'void main() {\n' +
9     '  gl_Position = u_ProjMatrix * a_Position;\n' +
10    '  v_Color = a_Color;\n' +
11    '}\n';
12
13 ...
23 function main() {
24   // Получить ссылку на элемент <canvas>
25   var canvas = document.getElementById('webgl');
26   // Получить ссылку на элемент с атрибутом id = nearFar
27   var nf = document.getElementById('nearFar');
28   ...
52   // Получить ссылку на переменную u_ProjMatrix
53   var u_ProjMatrix = gl.getUniformLocation(gl.program, 'u_ProjMatrix');
54   ...
59   // Создать матрицу для установки местоположения точки наблюдения
60   var projMatrix = new Matrix4();
61   // Зарегистрировать обработчик событий нажатий на клавиши
62   document.onkeydown = function(ev) { keydown(ev, gl, n, u_ProjMatrix,
63                                             projMatrix, nf); };
64
65   draw(gl, n, u_ProjMatrix, projMatrix, nf); // Нарисовать треугольники
66 }
67
68 ...
116 // Расстояния до ближней и дальней плоскостей отсечения
117 var g_near = 0.0, g_far = 0.5;
118 function keydown(ev, gl, n, u_ProjMatrix, projMatrix, nf) {
119   switch(ev.keyCode) {
120     case 39: g_near += 0.01; break; // Нажата клавиша со стрелкой вправо
121     case 37: g_near -= 0.01; break; // Нажата клавиша со стрелкой влево
122     case 38: g_far += 0.01; break; // Нажата клавиша со стрелкой вверх
123     case 40: g_far -= 0.01; break; // Нажата клавиша со стрелкой вниз
124     default: return; // Предотвратить ненужное перерисовывание
125   }
126 }

```

```

127 draw(gl, n, u_ProjMatrix, projMatrix, nf);
128 }
129
130 function draw(gl, n, u_ProjMatrix, projMatrix, nf) {
131     // Установить видимый объем, используя матрицу
132     projMatrix.setOrtho(-1, 1, -1, 1, g_near, g_far);
133
134     // Сохранить матрицу проекции в переменной u_ProjMatrix
135     gl.uniformMatrix4fv(u_ProjMatrix, false, projMatrix.elements);
136
137     gl.clear(gl.COLOR_BUFFER_BIT); // Очистить <canvas>
138
139     // Вывести текущие значения near и far
140     nf.innerHTML = 'near: ' + Math.round(g_near * 100)/100 + ', far: ' +
        ➡ Math.round(g_far*100)/100;
141
142     gl.drawArrays(gl.TRIANGLES, 0, n); // Нарисовать треугольники
143 }

```

Так же, как в программе `LookAtTrianglesWithKeys`, при нажатии на клавишу вызывается функция `keydown()`, определение которой начинается в строке 118. Эта функция в свою очередь вызывает функцию `draw()` (строка 127). Определение функции `draw()` начинается в строке 130. Она устанавливает границы видимого объема, выводит значения *near* и *far*, и затем рисует три треугольника. Ключевой особенностью этой программы является функция `draw()`; однако, прежде чем перейти к исследованию этой функции, рассмотрим сначала, как изменить содержимое HTML-элемента из JavaScript.

Изменение содержимого HTML-элемента из JavaScript

Способ изменения содержимого HTML-элемента из JavaScript напоминает способ рисования в элементе `<canvas>` из WebGL. То есть, после получения ссылки на HTML-элемент по его атрибуту `id` вызовом `getElementById()`, можно легко и просто записать текстовое сообщение в этот элемент.

В данной программе мы изменяем следующий элемент `<p>`, записывая в него сообщение, такое как «near: 0.0, far: 0.5»:

```

12     <p id="nearFar"> The near and far values are displayed here. </p>

```

Ссылка на элемент извлекается в строке 27, вызовом `getElementById()`, как описывалось выше. Чтобы получить ссылку на нужный элемент `<p>`, в вызов метода следует передать строку (`'nearFar'`), присвоенную атрибуту `id` элемента в строке 12, в файле HTML:

```

26 // Получить ссылку на элемент с атрибутом id = nearFar
27 var nf = document.getElementById('nearFar');

```

После сохранения ссылки на элемент `<p>` в переменной `nf` (в действительности `nf` является объектом JavaScript) остается просто изменить содержимое этого

элемента. Это легко можно сделать с помощью свойства `innerHTML` объекта. Например, если записать:

```
nf.innerHTML = 'Good Morning, Marisuke-san!';
```

На странице появится сообщение «Good Morning, Marisuke-san!» («Доброе утро, Марисуки-сан!»). В текст сообщения можно также вставлять теги HTML. Например, если свойству `innerHTML` присвоить строку «Good Morning, **Marisuke** -san!», слово «Marisuke» будет выделено жирным.

В программе `OrthoView.js` мы использовали следующее выражение для вывода текущих значений *near* и *far*, которые хранятся в глобальных переменных `g_near` и `g_far`, объявленных в строке 117. При формировании текста сообщения, эти значения форматируются с помощью `Math.round()`, как показано ниже:

```
139 // Вывести текущие значения near и far
140 nf.innerHTML = 'near: ' + Math.round(g_near * 100)/100 + ', far: ' +
    ➤ Math.round(g_far*100)/100;
```

Вершинный шейдер

Как показано в следующем фрагменте, вершинный шейдер почти не изменился, в сравнении с программой `LookAtRotatedTriangles.js`, за исключением изменившегося имени `uniform`-переменной (`u_ProjMatrix`) в строке 6. Эта переменная хранит матрицу, которая используется для настройки видимого объема. Соответственно, чтобы установить границы видимого объема, нам остается только умножить матрицу (`u_ProjMatrix`) на координаты вершины, как показано в строке 9:

```
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
    ...
6 'uniform mat4 u_ProjMatrix;\n' +
7 'varying vec4 v_Color;\n' +
8 'void main() {\n' +
9 '   gl_Position = u_ProjMatrix * a_Position;\n' +
10 '   v_Color = a_Color;\n' +
11 '}\n';
```

В строке 62 выполняется регистрация обработчика событий нажатий на клавиши со стрелками. Обратите внимание, что в последнем аргументе обработчику передается `nf` – ссылка на элемент `<p>`. Исходя из нажатой клавиши, обработчик устанавливает значения переменных `g_near` и `g_far`, которые затем выводятся в элементе `<p>` функцией `draw()`, вызываемой тут же:

```
61 // Зарегистрировать обработчик событий нажатий на клавиши
62 document.onkeydown = function(ev) { keydown(ev, gl, n, u_ProjMatrix,
    ➤projMatrix, nf); };
```

Функция `keydown()` (строка 118) выясняет, какая клавиша со стрелкой была нажата, и изменяет значение `g_near` и `g_far` перед вызовом `draw()` в строке 127.

Переменные `g_near` и `g_far`, которые используются в вызове метода `setOrtho()`, определяются в строке 117 как глобальные, потому что используются в двух функциях, `keydown()` и `draw()`:

```

116 // Расстояния до ближней и дальней плоскостей отсечения
117 var g_near = 0.0, g_far = 0.5;
118 function keydown(ev, gl, n, u_ProjMatrix, projMatrix, nf) {
119     switch(ev.keyCode) {
120         case 39: g_near += 0.01; break; // Нажата клавиша со стрелкой вправо
121         ...
122         case 40: g_far -= 0.01; break; // Нажата клавиша со стрелкой вниз
123         default: return; // Предотвратить ненужное перерисовывание
124     }
125 }
126
127 draw(gl, n, u_ProjMatrix, projMatrix, nf);
128 }
```

Теперь перейдем к функции `draw()` (строка 130). Порядок выполнения операций в функции `draw()`, остался прежним, как в программе `LookAtTrianglesWithKeys.js`, за исключением появившейся инструкции, изменяющей текст сообщения на странице (строка 140):

```

130 function draw(gl, n, u_ProjMatrix, projMatrix, nf) {
131     // Установить видимый объем, используя матрицу
132     projMatrix.setOrtho(-1, 1, -1, 1, g_near, g_far);
133
134     // Сохранить матрицу проекции в переменной u_ProjMatrix
135     gl.uniformMatrix4fv(u_ProjMatrix, false, projMatrix.elements);
136
137     gl.clear(gl.COLOR_BUFFER_BIT); // Очистить <canvas>
138
139     // Вывести текущие значения near и far
140     nf.innerHTML = 'near: ' + Math.round(g_near * 100)/100 + ', far: ' +
141                    'Math.round(g_far*100)/100;
142
143     gl.drawArrays(gl.TRIANGLES, 0, n); // Нарисовать треугольники
144 }
```

В строке 132 вычисляется матрица, представляющая видимый объем (`projMatrix`), которая затем передается в переменную `u_ProjMatrix`, в строке 135. В строке 140 выводятся текущие значения *near* и *far*. Наконец, в строке 142, выполняется рисование треугольников.

Изменение *near* или *far*

Если запустить программу и увеличить значение *near* (клавишей со стрелкой вправо), изображение изменится, как показано на рис. 7.14.

По умолчанию расстояние *near* равно 0.0, поэтому видны все три треугольника. Далее, после увеличения расстояния *near* нажатием клавиши со стрелкой вправо, синий треугольник (ближайший) исчез, потому что видимый объем сократился и треугольник оказался за его границами, как показано на рис. 7.15. Это состояние показано на рис. 7.14 в середине.

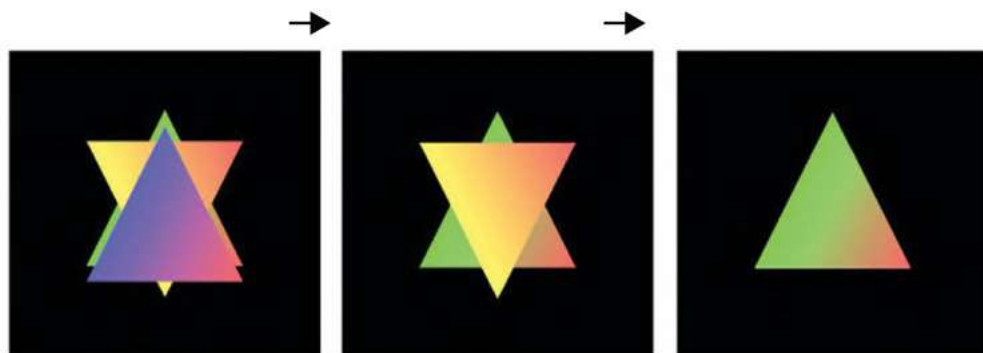


Рис. 7.14. Результат увеличения *near* клавишей со стрелкой вправо

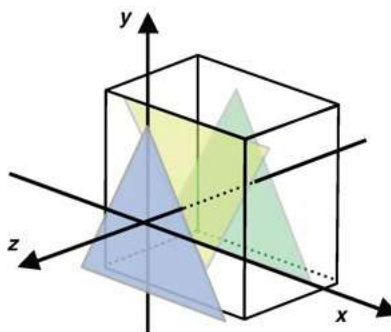


Рис. 7.15. Синий треугольник оказался за границами видимого объема

Если дальше продолжать увеличивать значение *near* нажатием клавиши со стрелкой вправо, до момента, когда оно превысит 0.2, ближняя плоскость отсечения окажется дальше желтого треугольника, соответственно треугольник окажется за границами видимого объема и исчезнет. На экране останется только зеленый треугольник (справа на рис. 7.14). Если после этого уменьшить значение *near* нажатием клавиши со стрелкой влево до значения меньше 0.2, желтый треугольник вновь появится на экране. Напротив, если продолжать увеличивать *near*, зеленый треугольник также исчезнет и на экране останется черная, пустая область рисования.

Как вы уже наверняка догадались, попытки перемещения дальней плоскости отсечения приводят к похожим результатам. Как показано на рис. 7.16, когда расстояние *far* окажется меньше 0.4, дальний (зеленый) треугольник исчезнет. Если продолжать уменьшать расстояние *far*, исчезнет желтый треугольник и на экране останется только синий.

Этот пример должен помочь вам уяснить роль видимого объема. Любой объект, который вы пожелаете отобразить, должен находиться в границах видимого объема.

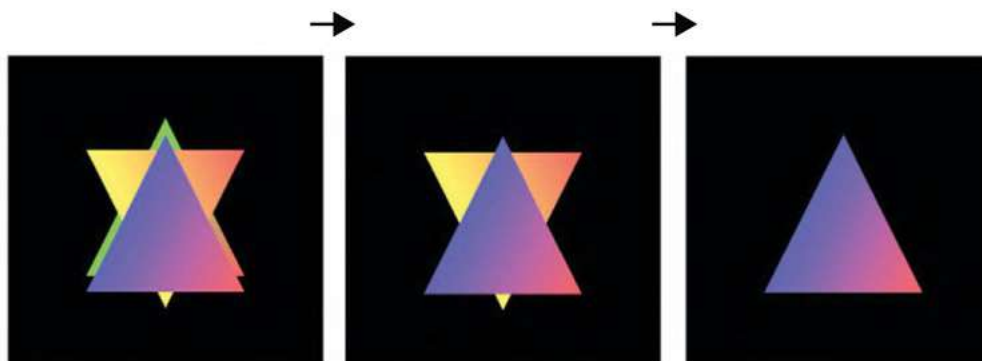


Рис. 7.16. Результат уменьшения far клавишей со стрелкой вниз

Восстановление отсеченных частей треугольников (LookAtTrianglesWithKeys_ViewVolume.js)

Если в программе `LookAtTrianglesWithKeys` продолжать нажимать клавиши со стрелками, в конечном итоге наступит момент, когда часть треугольника будет обрезана, как показано на рис. 7.17. Из предыдущего обсуждения становится очевидно, что этот эффект обусловлен выходом некоторой части сцены за границы видимого объема. В этом разделе мы изменим программу так, что она будет правильно отображать треугольники, устанавливая границы видимого объема как это необходимо.



Рис. 7.17. Часть треугольника оказалась обрезана

Как показано на рис. 7.17, дальний от точки наблюдения угол треугольника оказался обрезан. Совершенно понятно, что дальняя плоскость отсечения рас-

положена слишком близко к точке наблюдения, поэтому ее необходимо немного отодвинуть. Для этого мы изменим параметры, определяющие видимый объем: $left = -1.0$, $right = 1.0$, $bottom = -1.0$, $top = 1.0$, $near = 0.0$ и $far = 2.0$.

В новой программе будут использоваться две матрицы: матрица, определяющая границы видимого объема (матрица ортогональной проекции), и матрица, определяющая местоположение точки наблюдения (матрица вида). Так как `setOrtho()` устанавливает видимый объем, опираясь на местоположение точки наблюдения, необходимо сначала определить это местоположение, и только потом – видимый объем. Соответственно, мы сначала будем умножать матрицу вида на координаты вершины, чтобы получить координаты «видимые с указанной позиции», а затем – матрицу ортогональной проекции на вновь полученные координаты. Вычисления будут выполняться в соответствии с формулой 7.3.

Формула 7.3

<матрица ортогональной проекции> × <матрица вида> × <координаты вершины>

Эту формулу можно реализовать в вершинном шейдере, как показано в листинге 7.7.

Листинг 7.7. LookAtTrianglesWithKeys_ViewVolume.js

```
1 // LookAtTrianglesWithKeys_ViewVolume.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4 'attribute vec4 a_Position;\n' +
5 'attribute vec4 a_Color;\n' +
6 'uniform mat4 u_ViewMatrix;\n' +
7 'uniform mat4 u_ProjMatrix;\n' +
8 'varying vec4 v_Color;\n' +
9 'void main() {\n' +
10 '   gl_Position = u_ProjMatrix * u_ViewMatrix * a_Position;\n' +
11 '   v_Color = a_Color;\n' +
12 '}\n';
13 ...
24 function main() {
25     ...
51     // Получить ссылки на u_ViewMatrix и u_ProjMatrix
52     var u_ViewMatrix = gl.getUniformLocation(gl.program, 'u_ViewMatrix');
53     var u_ProjMatrix = gl.getUniformLocation(gl.program, 'u_ProjMatrix');
54     ...
59     // Создать матрицу для хранения матрицы вида
60     var viewMatrix = new Matrix4();
61     // Зарегистрировать обработчик событий нажатий клавиш
62     document.onkeydown = function(ev) { keydown(ev, gl, n, u_ViewMatrix,
63                                         viewMatrix); };
64
65     // Создать матрицу видимого объема и передать ее в u_ProjMatrix
66     var projMatrix = new Matrix4();
67     projMatrix.setOrtho(-1.0, 1.0, -1.0, 1.0, 0.0, 2.0);
68     gl.uniformMatrix4fv(u_ProjMatrix, false, projMatrix.elements);
```

```
68  
69 draw(gl, n, u_ViewMatrix, viewMatrix); // Draw the triangles  
70 }
```

В строке 66 производится вычисление матрицы ортогональной проекции (`projMatrix`) со значением расстояния *far*, измененным с 1.0 на 2.0. Полученная матрица передается в вершинный шейдер через переменную `u_ProjMatrix` (строка 67). Здесь используется `uniform`-переменная, потому что элементы матрицы одинаковы для всех вершин. Если запустить эту программу и попробовать сместить точку наблюдения, как мы делали это выше, можно убедиться, что треугольники больше не обрезаются (см. рис. 7.18).



Рис. 7.18. LookAtTrianglesWithKeys_ViewVolume

Эксперименты с примером программы

Как отмечалось в разделе «Определение видимого объема», если соотношение сторон ближней плоскости отсечения будет отличаться от соотношения сторон области рисования элемента `<canvas>`, WebGL произведет масштабирование и формы геометрических фигур в сцене будут искажены. Давайте исследуем этот эффект. Для начала в программе `OrthoView_halfSize` (основанной на листинге 7.7) мы уменьшим текущие размеры ближней плоскости отсечения вдвое, сохранив соотношение размеров сторон:

```
projMatrix.setOrtho(-0.5, 0.5, -0.5, 0.5, 0, 0.5);
```

Результат показан на рис. 7.19 слева. Как видите, размеры треугольников увеличились вдвое, по сравнению с предыдущим примером, потому что размеры области рисования `<canvas>` остались теми же, что и прежде. Обратите внимание, что части треугольников, вышедшие за границы ближней плоскости отсечения, оказались обрезаны.

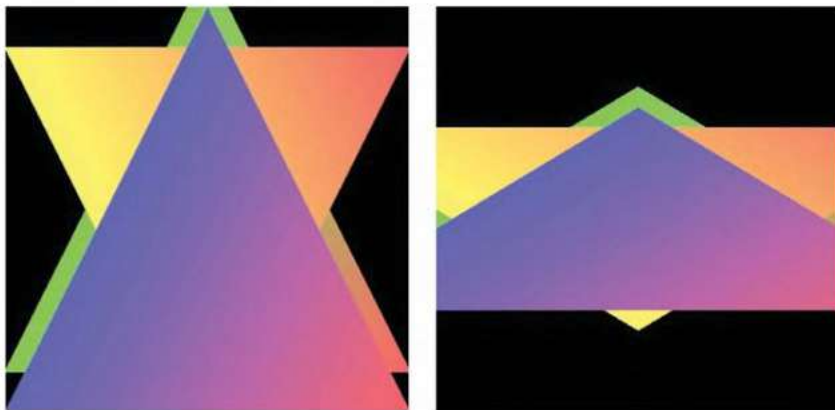


Рис. 7.19. Результаты изменения размеров ближней плоскости отсечения

В программе `OrthoView_halfWidth` мы уменьшили только ширину ближней плоскости отсечения, изменив первые два аргумента в вызове `setOrtho()`, как показано ниже:

```
projMatrix.setOrtho(-0.3, 0.3, -1.0, 1.0, 0.0, 0.5);
```

Результат можно увидеть на рис. 7.19 справа. Горизонтальный размер ближней плоскости отсечения оказался меньше горизонтального размера области рисования `<canvas>`, поэтому она была «растянута», что и вызвало такие искажения.

Определение видимого объема в форме четырехгранной пирамиды

На рис. 7.20 изображена дорога, обсаженная деревьями. Все деревья на этом рисунке имеют примерно одинаковую высоту, но деревья, расположенные дальше от наблюдателя, кажутся меньше. Точно так же здание, видимое на заднем плане, выглядит ниже деревьев, расположенных ближе к наблюдателю, хотя в действительности это не так. Такой эффект, когда удаленные объекты выглядят меньше, создает ощущение глубины. Интересно отметить, что хотя наши глаза видят окружающий мир именно так, на детских рисунках редко можно увидеть подобную перспективу.

Когда видимый объем определяется как прямоугольный параллелепипед, треугольники одинакового размера рисуются одинаковыми, независимо от их расстояния до точки наблюдения. Чтобы исправить этот недостаток, можно определить видимый объем в форме четырехгранной пирамиды, что позволит вызвать ощущение глубины, как на рис. 7.20.

В этом разделе мы напомним программу `PerspectiveView`, в которой определим видимый объем в форме четырехгранной пирамиды, простирающейся вдоль оси Z в сторону отрицательных значений с вершиной в точке наблюдения с коор-

динатами $(0, 0, 5)$. На рис. 7.21 показан скриншот программы PerspectiveView и расположение треугольников.



Рис. 7.20. Дорога, обсаженная деревьями

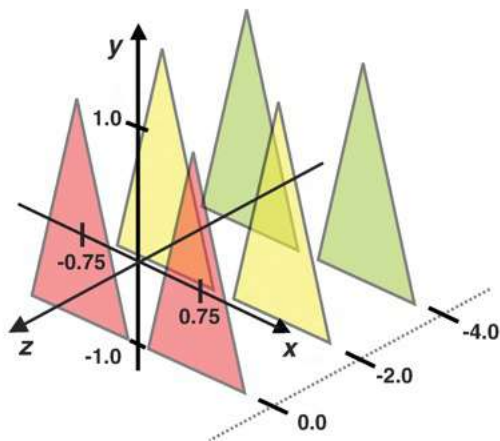


Рис. 7.21. PerspectiveView; расположение треугольников

Как видно на рис. 7.21 справа, три треугольника с одинаковыми размерами расположены в два ряда, справа и слева, вдоль оси Z, подобно деревьям вдоль дороги на рис. 7.20. После установки видимого объема в форме четырехгранной пирамиды WebGL автоматически будет уменьшать более далекие объекты, создавая эффект глубины, как показано на рис. 7.21 слева.

Чтобы действительно заметить разницу в размерах, как в реальном мире, объекты должны находиться на существенном расстоянии. Например, чтобы дальняя грань параллелепипеда выглядела меньше передней, этот параллелепипед должен иметь существенную длину. Поэтому на сей раз мы будем использовать несколько большее расстояние $(0, 0, 0.5)$ от точки наблюдения (прежде точка наблюдения устанавливалась в начало координат $(0, 0, 0)$).

Определение границ видимого объема в форме четырехгранной пирамиды

На рис. 7.22 показано, как выглядит видимый объем в форме четырехгранной пирамиды. Как и в случае с видимым объемом в форме прямоугольного параллелепипеда, границы видимого объема в форме четырехгранной пирамиды определяются местоположением точки наблюдения, а также ближней и дальней плоскостями отсечения. Объекты, не попавшие в видимый объем, не отображаются, а для объектов на границе отображаются только те их части, которые попали в видимый объем.

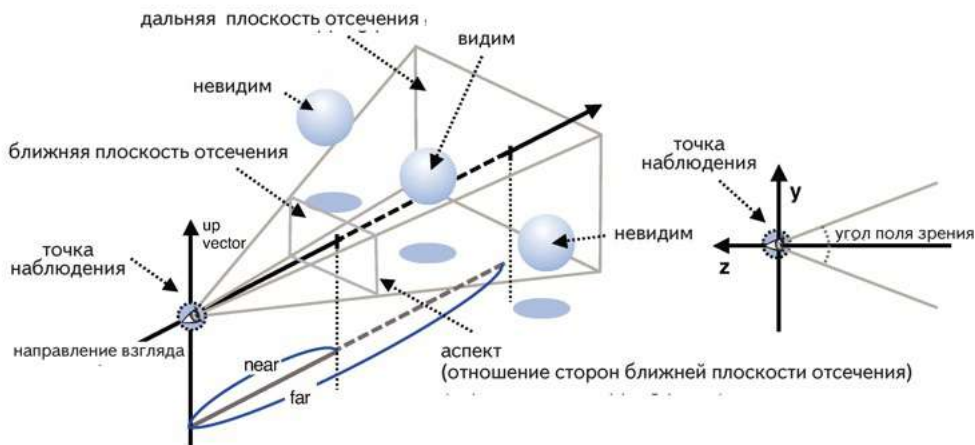


Рис. 7.22. Видимый объем в форме четырехгранной пирамиды

Независимо от формы видимого объема, его границы всегда задаются с помощью матриц, правда аргументы методов, вычисляющих эти матрицы, отличаются. Для вычисления матрицы, описывающей видимый объем в форме четырехгранной пирамиды, используется метод `setPerspective()` объекта `Matrix4`.

`Matrix4.setPerspective(fov, aspect, near, far)`

Вычисляет матрицу (матрицу перспективной проекции), которая определяет видимый объем, исходя из своих аргументов, и сохраняет ее в объекте `Matrix4`. При этом значение `near` должно быть меньше значения `far`.

Параметры	fov	Определяет величину угла поля зрения, образованного верхней и нижней кромками ближней плоскости отсечения. Этот параметр должен иметь значение больше 0.
	aspect	Определяет соотношение сторон. ближней плоскости отсечения (ширина/высота).
	near, far	Определяют расстояние до ближней и дальней плоскости отсечения от точки наблюдения вдоль линии взгляда ($near > 0$ и $far > 0$).
Возвращаемое значение	нет	

Матрица, описывающая видимый объем в форме четырехгранной пирамиды, называется **матрицей перспективной проекции** (perspective projection matrix).

Обратите внимание, что определение ближней плоскости отсечения отличается от ее определения в видимом объеме в форме параллелепипеда: второй аргумент – `aspect` – представляет отношение сторон ближней плоскости отсечения. Например, если принять высоту равной 100, а ширину равной 200, отношение (`aspect`) будет равно 0.5.

На рис. 7.23 показано расположение треугольников в используемом видимом объеме. Этот видимый объем имеет следующие характеристики: $near = 1.0$, $far = 100$, $aspect = 1.0$ (соответствует отношению сторон области рисования `<canvas>`) и $fov = 30.0$.

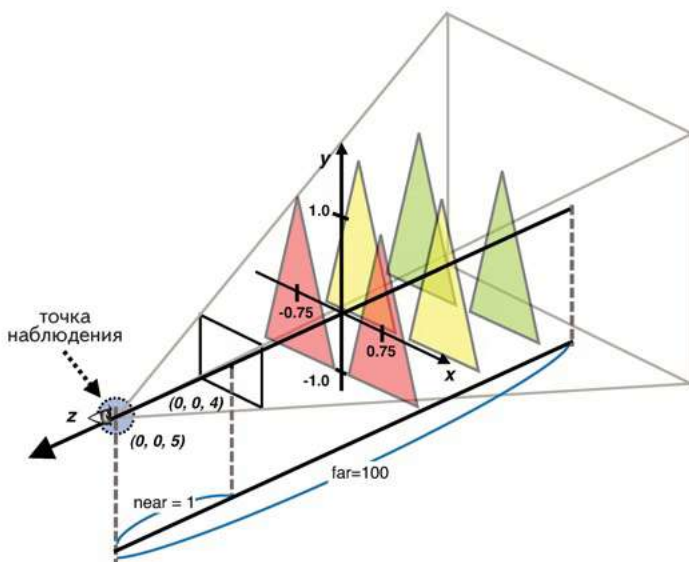


Рис. 7.23. Расположение треугольников в видимом объеме, имеющем форму четырехгранной пирамиды

Своей структурой новая программа напоминает программу `LookAtTrianglesWithKeys_ViewVolume.js` из предыдущего раздела. Рассмотрим ее.

Пример программы (PerspectiveView.js)

В листинге 7.8 приводится исходный код программы PerspectiveView.js.

Листинг 7.8. PerspectiveView.js

```
1 // PerspectiveView.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'attribute vec4 a_Color;\n' +
6   'uniform mat4 u_ViewMatrix;\n' +
7   'uniform mat4 u_ProjMatrix;\n' +
8   'varying vec4 v_Color;\n' +
9   'void main() {\n' +
10    '  gl_Position = u_ProjMatrix * u_ViewMatrix * a_Position;\n' +
11    '  v_Color = a_Color;\n' +
12    '}\n';
13 ...
24 function main() {
14   ...
41   // Определить координаты вершин и цвет (синий треугольник - первый)
42   var n = initVertexBuffers(gl);
15   ...
51   // Получить ссылки на переменные u_ViewMatrix и u_ProjMatrix
52   var u_ViewMatrix = gl.getUniformLocation(gl.program, 'u_ViewMatrix');
53   var u_ProjMatrix = gl.getUniformLocation(gl.program, 'u_ProjMatrix');
16   ...
59   var viewMatrix = new Matrix4(); // Матрица вида
60   var projMatrix = new Matrix4(); // Матрица проекции
61
62   // Вычислить матрицы вида и проекции
63   viewMatrix.setLookAt(0, 0, 5, 0, 0, -100, 0, 1, 0);
64   projMatrix.setPerspective(30, canvas.width/canvas.height, 1, 100);
65   // Передать матрицы в переменные u_ViewMatrix и u_ProjMatrix
66   gl.uniformMatrix4fv(u_ViewMatrix, false, viewMatrix.elements);
67   gl.uniformMatrix4fv(u_ProjMatrix, false, projMatrix.elements);
17   ...
72   // Нарисовать треугольники
73   gl.drawArrays(gl.TRIANGLES, 0, n);
74 }
75
76 function initVertexBuffers(gl) {
77   var verticesColors = new Float32Array([
78     // Три треугольника справа
79     0.75, 1.0, -4.0, 0.4, 1.0, 0.4, // Дальний зеленый треугольник
80     0.25, -1.0, -4.0, 0.4, 1.0, 0.4,
81     1.25, -1.0, -4.0, 1.0, 0.4, 0.4,
82
83     0.75, 1.0, -2.0, 1.0, 1.0, 0.4, // Желтый треугольник в середине
84     0.25, -1.0, -2.0, 1.0, 1.0, 0.4,
85     1.25, -1.0, -2.0, 1.0, 0.4, 0.4,
86
87     0.75, 1.0, 0.0, 0.4, 0.4, 1.0, // Ближний синий треугольник
```

```

88     0.25, -1.0, 0.0, 0.4, 0.4, 1.0,
89     1.25, -1.0, 0.0, 1.0, 0.4, 0.4,
90
91     // Три треугольника слева
92     -0.75, 1.0, -4.0, 0.4, 1.0, 0.4, // Дальний зеленый треугольник
93     -1.25, -1.0, -4.0, 0.4, 1.0, 0.4,
94     -0.25, -1.0, -4.0, 1.0, 0.4, 0.4,
95
96     -0.75, 1.0, -2.0, 1.0, 1.0, 0.4, // Желтый треугольник в середине
97     -1.25, -1.0, -2.0, 1.0, 1.0, 0.4,
98     -0.25, -1.0, -2.0, 1.0, 0.4, 0.4,
99
100    -0.75, 1.0, 0.0, 0.4, 0.4, 1.0, // Ближний синий треугольник
101    -1.25, -1.0, 0.0, 0.4, 0.4, 1.0,
102    -0.25, -1.0, 0.0, 1.0, 0.4, 0.4,
103    });
104    var n = 18; // По три вершины на треугольник * 6
105    ...
138    return n;
139 }

```

Вершинный и фрагментный шейдеры полностью идентичны (включая имена переменных) шейдерам в программе `LookAtTriangles_ViewVolume.js`.

Порядок выполнения операций в JavaScript-функции `main()` также остался прежним. В результате вызова функции `initVertexBuffers()` в строке 42 в буферный объект записываются координаты вершин и цвета треугольников. Функция `initVertexBuffers()` определяет вершины и цвет шести треугольников: три треугольника справа (строки с 79 по 89) и три треугольника слева (строки с 92 по 102). Соответственно, число вершин (строка 104) теперь стало равно 18 ($3 \times 6 = 18$, по три вершины на каждый из шести треугольников).

В строках 52 и 53 программа получает ссылки на `uniform`-переменные, где должны храниться матрица вида и матрица перспективной проекции. Затем, в строках 59 и 60, в этих переменных сохраняются созданные матрицы. В строке 63 вычисляется матрица вида, с учетом позиции точки наблюдения (0, 0, 5), направления взгляда вдоль оси Z в сторону отрицательных значений и направления вверх вдоль оси Y, в сторону положительных значений. Наконец, в строке 64 определяется матрица проекции, описывающая видимый объем в форме четырехгранной пирамиды:

```

64    projMatrix.setPerspective(30, canvas.width/canvas.height, 1, 100);

```

Второй аргумент, `aspect` (отношение высоты ближней плоскости отсечения к ее ширине) определяется как отношение высоты к ширине элемента `<canvas>` (свойства `height` и `width`), поэтому никакие изменения размеров элемента `<canvas>` не приведут к искажению отображаемых объектов.

Далее, после вычисления матриц вида и проекции, они передаются в соответствующие `uniform`-переменные (строки 66 и 67). Наконец, в строке 73 осуществляется рисование треугольников и на экране появляется перспективное изображение, напоминающее рис. 7.20.

Выше мы затронули один интересный вопрос, но не стали развивать его: почему для определения границ видимого объема используются матрицы? Теперь мы коротко обсудим его, не углубляясь в математический аппарат.

Назначение матрицы проекции

Начнем с исследования матрицы перспективной проекции. Взглянув на скриншот программы `PerspectiveView` (рис. 7.24), можно заметить, что после применения матрицы проекции, удаленные объекты претерпели два изменения.

Во-первых, чем дальше от точки наблюдения отстоит треугольник, тем более маленьким он выглядит. Во-вторых, изменились видимые позиции треугольников, и теперь кажется, что они смещены внутрь сцены, ближе к линии взгляда. В сравнении с треугольниками одинакового размера, изображенными на рис. 7.25 слева, в перспективной проекции были применены два преобразования: (1) более далекие треугольники уменьшены в размерах с масштабным коэффициентом, пропорциональным расстоянию от наблюдателя, и (2) смещены в сторону линии взгляда (на рис. 7.25 справа). Эти два преобразования создают эффект, который наблюдается на фотографии, изображенной на рис. 7.20.

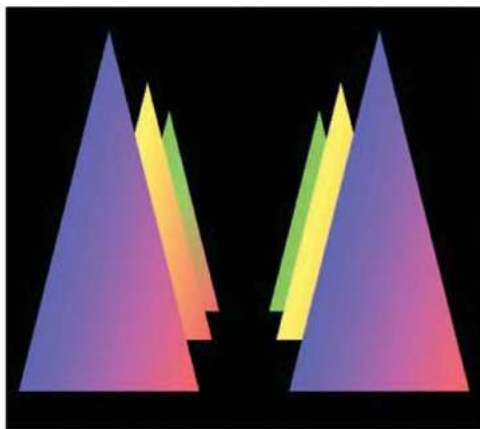


Рис. 7.24. `PerspectiveView`

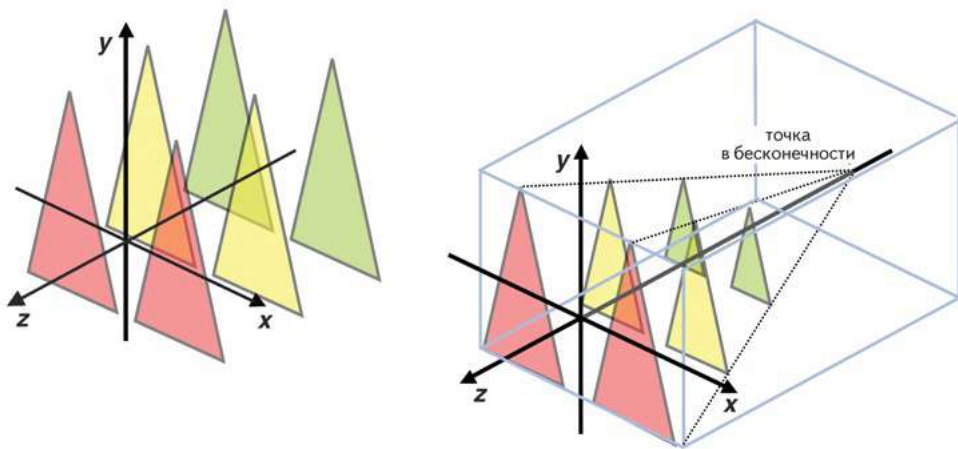


Рис. 7.25. Суть преобразований перспективной проекции

Из этого следует, что определение видимого объема можно выразить, как комбинацию преобразований, таких как масштабирование или трансляцию геоме-

трических фигур и объектов, в соответствии с формой видимого объема. Метод `setPerspective()` объекта `Matrix4` автоматически вычисляет матрицу преобразования, исходя из своих аргументов, описывающих видимый объем. Подробнее об элементах этой матрицы рассказывается в приложении С, «Матрицы проекций». Если вам интересен математический аппарат преобразований координат в зависимости от формы видимого объема, обращайтесь к книге «Computer Graphics».

Если взглянуть с другой стороны, преобразования, связанные с перспективной проекцией, трансформируют видимый объем в форме четырехгранной пирамиды в видимый объем в форме прямоугольного параллелепипеда (рис. 7.25 справа).

Обратите внимание, что матрица ортогональной проекции не содержит всего, что требуется для преобразований, вызывающих требуемый визуальный эффект. Скорее она содержит лишь предварительно подготовленные исходные данные, которые требуется передать в вершинный шейдер, где и выполняются все необходимые операции. Любопытствующих мы отсылаем к приложению D, «WebGL/OpenGL: лево- или правосторонняя система координат?».

Все матрицы вместе – проекции, модели и вида – содержат все, что нужно, чтобы выполнить геометрические преобразования (трансляцию, вращение, масштабирование), необходимые для достижения разных визуальных эффектов. В следующем разделе мы узнаем, как комбинировать эти матрицы, на простом примере.

Использование всех матриц (модели, вида и проекции)

Одна из проблем, проявившихся в программе `PerspectiveView.js`, – большой объем кода, необходимого для определения координат вершин и цветов. Пока мы имеем дело всего с шестью треугольниками, поэтому проблема остается под контролем, но с увеличением числа треугольников ситуация будет становиться все хуже.

К счастью, существует эффективный прием устранения этой проблемы. Если внимательно рассмотреть изображение на рис. 7.25 слева, можно заметить, что конфигурация сцены идентична представленной на рис. 7.26, где треугольники, изображенные пунктиром, смещены вдоль оси X в положительном (0.75) и отрицательном (–0.75) направлениях, соответственно.

Таким образом, мы могли бы нарисовать треугольники в программе `PerspectiveView`, используя следующий алгоритм:

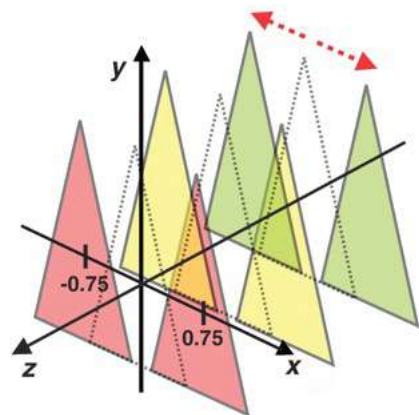


Рис. 7.26. Рисование треугольников выполняется после трансляции (смещения)

1. Подготовить координаты вершин для трех треугольников, расположенных по центру, вдоль оси Z.
2. Сместить исходные треугольники на 0.75 единицы вдоль оси X и нарисовать их.
3. Сместить исходные треугольники на -0.75 единицы вдоль оси X и нарисовать их.

Попробуем воплотить этот алгоритм в программный код (`PerspectiveView_mvp`).

В первоначальной версии программы `PerspectiveView` матрицы проекции и вида использовались для определения позиции точки наблюдения (наблюдателя) и формы видимого объема. В программе `PerspectiveView_mvp` используется дополнительная матрица модели, описывающая трансляцию (перемещение) треугольников.

В настоящий момент имеет смысл рассмотреть, какие преобразования описывают эти матрицы. Для этого обратимся вновь к программе `LookAtTriangles`, написанной нами ранее, которая позволяет увидеть, как выглядит треугольник после поворота с определенной точки. В тот раз мы использовали выражение, идентичное формуле 7.1:

$$\langle \text{матрица вида} \rangle \times \langle \text{матрица модели} \rangle \times \langle \text{координаты вершины} \rangle$$

Опираясь на это выражение, мы написали программу `LookAtTriangles_ViewVolume`, которая устраняла эффект обрезания треугольника, и использовали в ней следующее выражение, в котором задействовали матрицу проекции для описания видимого объема в форме прямоугольного параллелепипеда или четырехгранной пирамиды, идентичное формуле 7.3:

$$\langle \text{матрица проекции} \rangle \times \langle \text{матрица вида} \rangle \times \langle \text{координаты вершины} \rangle$$

Из этих двух выражений легко вывести следующее:

Формула 7.4

$$\begin{aligned} \langle \text{матрица проекции} \rangle \times \langle \text{матрица вида} \rangle \times \langle \text{матрица модели} \rangle \\ \times \langle \text{координаты вершины} \rangle \end{aligned}$$

Формула 7.4 показывает, что система WebGL позволяет вычислять окончательные координаты вершин, используя матрицы трех типов: матрицу модели, матрицу вида и матрицу проекции.

Будет понятнее, если учесть, что формула 7.4 идентична формуле 7.1, в которой матрица проекции является единичной матрицей, и идентична формуле 7.3, в которой единичной является матрица модели. Как уже говорилось в главе 4, единичная матрица в операции умножения матриц ведет себя подобно 1 в умножении чисел. Умножение на единичную матрицу не оказывает никакого эффекта на результат.

А теперь напомним программу, реализовав в ней формулу 7.4.

Пример программы (PerspectiveView_mvp.js)

Исходный код программы PerspectiveView_mvp.js приводится в листинге 7.9. Основной порядок выполнения операций остался прежним, как в программе PerspectiveView.js. Единственное отличие – вычисления в вершинном шейдере (строка 11), реализующие формулу 7.4, и передача дополнительной матрицы (`u_ModelMatrix`), используемой в вычислениях.

Листинг 7.9. PerspectiveView_mvp.js

```

1 // PerspectiveView_mvp.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'attribute vec4 a_Color;\n' +
6   'uniform mat4 u_ModelMatrix;\n' +
7   'uniform mat4 u_ViewMatrix;\n' +
8   'uniform mat4 u_ProjMatrix;\n' +
9   'varying vec4 v_Color;\n' +
10  'void main() {\n' +
11    '  gl_Position = u_ProjMatrix * u_ViewMatrix * u_ModelMatrix * a_Position;\n' +
12    '  v_Color = a_Color;\n' +
13  '}\n';
14
15 ...
16
17 function main() {
18   ...
19   // Определить координаты вершин и цвет (синий треугольник - первый)
20   var n = initVertexBuffers(gl);
21   ...
22   // Получить ссылки на u_ModelMatrix, u_ViewMatrix и u_ProjMatrix
23   var u_ModelMatrix = gl.getUniformLocation(gl.program, 'u_ModelMatrix');
24   var u_ViewMatrix = gl.getUniformLocation(gl.program, 'u_ViewMatrix');
25   var u_ProjMatrix = gl.getUniformLocation(gl.program, 'u_ProjMatrix');
26   ...
27   var modelMatrix = new Matrix4(); // Матрица модели
28   var viewMatrix = new Matrix4(); // Матрица вида
29   var projMatrix = new Matrix4(); // Матрица проекции
30
31   // Вычислить матрицу модели, матрицу вида и матрицу проекции
32   modelMatrix.setTranslate(0.75, 0, 0); // Сместить на 0.75 единицы
33   viewMatrix.setLookAt(0, 0, 5, 0, 0, -100, 0, 1, 0);
34   projMatrix.setPerspective(30, canvas.width/canvas.height, 1, 100);
35   // Передать матрицы в uniform-переменные
36   gl.uniformMatrix4fv(u_ModelMatrix, false, modelMatrix.elements);
37   gl.uniformMatrix4fv(u_ViewMatrix, false, viewMatrix.elements);
38   gl.uniformMatrix4fv(u_ProjMatrix, false, projMatrix.elements);
39
40   gl.clear(gl.COLOR_BUFFER_BIT); // Очистить <canvas>
41
42   gl.drawArrays(gl.TRIANGLES, 0, n); // Нарисовать треугольники справа
43
44   // Подготовить матрицу модели для другой тройки треугольников
45   modelMatrix.setTranslate(-0.75, 0, 0); // Сместить на -0.75 единицы

```



```
80 // Изменить только матрицу модели
81 gl.uniformMatrix4fv(u_ModelMatrix, false, modelMatrix.elements);
82
83 gl.drawArrays(gl.TRIANGLES, 0, n); // Нарисовать треугольники слева
84 }
85
86 function initVertexBuffers(gl) {
87     var verticesColors = new Float32Array([
88         // Координаты вершин и цвета
89         0.0, 1.0, -4.0, 0.4, 1.0, 0.4, // Дальний зеленый треугольник
90         -0.5, -1.0, -4.0, 0.4, 1.0, 0.4,
91         0.5, -1.0, -4.0, 1.0, 0.4, 0.4,
92
93         0.0, 1.0, -2.0, 1.0, 1.0, 0.4, // Желтый треугольник в середине
94         -0.5, -1.0, -2.0, 1.0, 1.0, 0.4,
95         0.5, -1.0, -2.0, 1.0, 0.4, 0.4,
96
97         0.0, 1.0, 0.0, 0.4, 0.4, 1.0, // Ближний синий треугольник
98         -0.5, -1.0, 0.0, 0.4, 0.4, 1.0,
99         0.5, -1.0, 0.0, 1.0, 0.4, 0.4,
100    ]);
101    ...
102    return n;
103 }
```

На этот раз нам нужно передать в вершинный шейдер еще одну матрицу – матрицу модели. Поэтому мы добавили переменную `u_ModelMatrix` (строка 6). Сама матрица используется в строке 11, где реализована формула 7.5:

```
11 ' gl_Position = u_ProjMatrix * u_ViewMatrix * u_ModelMatrix * a_Position;\n' +
```

В строке 43 вызывается JavaScript-функция `initVertexBuffers()`. Она сохраняет координаты вершин треугольников в буферном объекте, который определен в строке 87. На этот раз мы имеем координаты трех треугольников, расположенных по центру вдоль оси Z, вместо шести, как в программе `PerspectiveView.js`. Как уже отмечалось выше, это обусловлено тем, что мы будем выполнять смещение сразу трех треугольников.

В строке 53 программа получает ссылку на переменную `u_ModelMatrix` в вершинном шейдере. В строке 61 создается сама матрица модели (`modelMatrix`), в строке 66 программа вычисляет матрицу на основе указанных нами аргументов. Данная матрица описывает трансляцию (смещение) треугольников на 0.75 единицы вдоль оси X:

```
65 // Вычислить матрицу модели, матрицу вида и матрицу проекции
66 modelMatrix.setTranslate(0.75, 0, 0); // Сместить на 0.75 единицы
...
70 gl.uniformMatrix4fv(u_ModelMatrix, false, modelMatrix.elements);
...
76 gl.drawArrays(gl.TRIANGLES, 0, n); // Нарисовать треугольники справа
```

Вычисление остальных матриц выполняется точно так же, как в программе `PerspectiveView.js`. В строке 70 готовая матрица модели передается в `uniform-`

переменную и затем в строке 76 выполняется рисование правого ряда треугольников.

Аналогично выполняется рисование левого ряда треугольников, смещенных на -0.75 единицы вдоль оси X, для этого матрица модели вычисляется заново в строке 79. Поскольку матрицы вида и проекции не изменились, нам требуется передать в `uniform`-переменную только матрицу модели (строка 81). После передачи матрицы выполняется вторая операция рисования вызовом `gl.drawArrays()` (строка 83):

```
78 // Подготовить матрицу модели для другой тройки треугольников
79 modelMatrix.setTranslate(-0.75, 0, 0); // Сместить на -0.75 единицы
80 // Изменить только матрицу модели
81 gl.uniformMatrix4fv(u_ModelMatrix, false, modelMatrix.elements);
82
83 gl.drawArrays(gl.TRIANGLES, 0, n); // Нарисовать треугольники слева
```

Как видите, данный прием позволил нарисовать два ряда треугольников, используя комплект данных для одного ряда, и уменьшить объем кода с определением вершин, но это привело к увеличению числа вызовов `gl.drawArrays()`. Разница в эффективности этих двух подходов зависит от конкретной реализации WebGL.

Эксперименты с примером программы

В программе `PerspectiveView_mvp` выражение *<матрица проекции>* $\times$ *<матрица вида>* $\times$ *<матрица модели>* вычисляется непосредственно внутри вершинного шейдера. Эти вычисления повторяются для каждой вершины, поэтому нет необходимости производить их в вершинном шейдере. Произведение можно вычислить заранее, в коде на JavaScript, как мы сделали это в программе `LookAtRotatedTriangles_mvMatrix`, и передавать в вершинный шейдер единственную матрицу. Эту матрицу можно назвать **матрицей модели вида проекции** (`model view projection matrix`), а переменной для ее передачи дать имя `u_MvpMatrix`. Программа, демонстрирующая этот подход, называется `ProjectiveView_mvMatrix`. В ней изменился вершинный шейдер и, как показано ниже, он стал намного проще:

```
1 // PerspectiveView_mvMatrix.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'attribute vec4 a_Color;\n' +
6   'uniform mat4 u_MvpMatrix;\n' +
7   'varying vec4 v_Color;\n' +
8   'void main() {\n' +
9   '   gl_Position = u_MvpMatrix * a_Position;\n' +
10  '   v_Color = a_Color;\n' +
11  '}\n';
```

В строке 51, в JavaScript-функции `main()` приобретается ссылка на `uniform`-переменную `u_MvpMatrix`, затем в строке 57 в ней создается матрица:

```
50 // Получить ссылку на переменную u_MvpMatrix
51 var u_MvpMatrix = gl.getUniformLocation(gl.program, 'u_MvpMatrix');
...
57 var modelMatrix = new Matrix4(); // Матрица модели
58 var viewMatrix = new Matrix4(); // Матрица вида
59 var projMatrix = new Matrix4(); // Матрица проекции
60 var mvpMatrix = new Matrix4(); // Матрица модели вида проекции
61
62 // Вычислить матрицы модели, вида и проекции
63 modelMatrix.setTranslate(0.75, 0, 0);
64 viewMatrix.setLookAt(0, 0, 5, 0, 0, -100, 0, 1, 0);
65 projMatrix.setPerspective(30, canvas.width/canvas.height, 1, 100);
66 // Вычислить матрицу модели вида проекции
67 mvpMatrix.set(projMatrix).multiply(viewMatrix).multiply(modelMatrix);
68 // Передать матрицу модели вида проекции в u_MvpMatrix
69 gl.uniformMatrix4fv(u_MvpMatrix, false, mvpMatrix.elements);
...
73 gl.drawArrays(gl.TRIANGLES, 0, n); // Нарисовать треугольники
74
75 // Подготовить матрицу модели для другого ряда треугольников
76 modelMatrix.setTranslate(-0.75, 0, 0);
77 // Вычислить матрицу модели вида проекции
78 mvpMatrix.set(projMatrix).multiply(viewMatrix).multiply(modelMatrix);
79 // Передать матрицу модели вида проекции в u_MvpMatrix
80 gl.uniformMatrix4fv(u_MvpMatrix, false, mvpMatrix.elements);
81
82 gl.drawArrays(gl.TRIANGLES, 0, n); // Нарисовать треугольники
83 }
```

Самые важные вычисления производятся в строке 67. Сначала матрица проекции (`projMatrix`) присваивается переменной `mvpMatrix`, затем умножается на матрицу вида (`viewMatrix`) и матрицу модели (`modelMatrix`). Результат сохраняется обратно в `mvpMatrix`, передается в переменную `u_MvpMatrix` (строка 69) и потом, в строке 73, выполняется рисование правого ряда треугольников. Аналогично вычисляется матрица вида модели проекции для левого ряда треугольников (строка 78). После этого, в строке 80, она передается в переменную `u_MvpMatrix` и в строке 82 выполняется рисование треугольников.

Теперь, получив все эти знания, вы сможете писать код, способный смещать точку наблюдения, определять форму видимого объема и позволяющий рассматривать трехмерные объекты под разными углами. Кроме того, вы теперь знаете, как решить проблему с обрезанием объектов, не попавших целиком в видимый объем. Однако, остается еще одна потенциальная проблема. При смещении точки наблюдения может возникнуть ситуация, когда объект переднего плана скроется за объектом заднего плана. Давайте посмотрим, как проявляется эта проблема.

Правильная обработка объектов переднего и заднего плана

В реальном мире, если положить на стол два кубика, как показано на рис. 7.27, ближний кубик частично закроет собой дальний кубик.

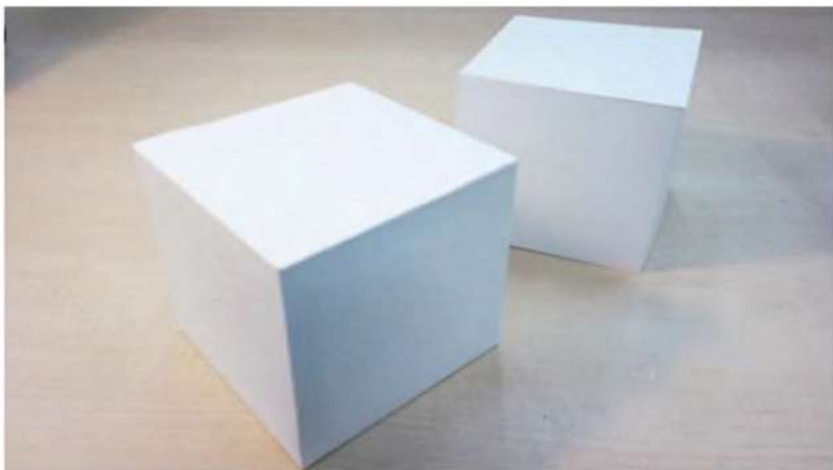


Рис. 7.27. Ближний кубик частично закрывает дальний

Если взглянуть на скриншоты программ, созданных нами к настоящему моменту, таких как `PerspectiveView` (см. рис. 7.21), можно заметить, что зеленый треугольник частично скрыт за желтым и синим треугольниками. Все выглядит так, будто система WebGL способна правильно отображать трехмерные объекты, с учетом их взаиморасположения в пространстве.

К сожалению это не так. По умолчанию, чтобы ускорить процесс рисования, WebGL рисует объекты в порядке следования вершин в буферном объекте. До сих пор мы всегда определяли вершины так, чтобы объекты заднего плана рисовались первыми, благодаря чему получали естественные изображения.

Например, в программе `PerspectiveView_mvMatrix.js` мы определили координаты и цвета треугольников в порядке, как показано ниже. Обратите внимание на координату *Z* (выделена жирным):

```
var verticesColors = new Float32Array([
  // Координаты вершин и цвета
  0.0,  1.0, -4.0, 0.4, 1.0, 0.4, // Дальний зеленый треугольник
  -0.5, -1.0, -4.0, 0.4, 1.0, 0.4,
  0.5, -1.0, -4.0, 1.0, 0.4, 0.4,

  0.0,  1.0, -2.0, 1.0, 1.0, 0.4, // Желтый треугольник в середине
  -0.5, -1.0, -2.0, 1.0, 1.0, 0.4,
  0.5, -1.0, -2.0, 1.0, 0.4, 0.4,

  0.0,  1.0, 0.0, 0.4, 0.4, 1.0, // Ближний синий треугольник
  -0.5, -1.0, 0.0, 0.4, 0.4, 1.0,
  0.5, -1.0, 0.0, 1.0, 0.4, 0.4,
]);
```

WebGL рисует треугольники в порядке уменьшения координаты *Z*, как мы определили в этом фрагменте (то есть, первым рисуется зеленый треугольник

[дальний], затем желтый [в середине] и, наконец, синий [ближний]). Это гарантирует, что объекты, расположенные ближе к точке наблюдения, будут перекрывать объекты, расположенные дальше, как показано на рис. 7.13.

Чтобы убедиться в этом, изменим порядок определения треугольников так, чтобы первым рисовался ближний синий треугольник, затем средний желтый и, наконец, зеленый дальний:

```
var verticesColors = new Float32Array([
  // Координаты вершин и цвета
  0.0,  1.0, 0.0, 0.4, 0.4, 1.0,  // Ближний синий треугольник
  -0.5, -1.0, 0.0, 0.4, 0.4, 1.0,
  0.5, -1.0, 0.0, 1.0, 0.4, 0.4

  0.0,  1.0, -2.0, 1.0, 1.0, 0.4,  // Желтый треугольник в середине
  -0.5, -1.0, -2.0, 1.0, 1.0, 0.4,
  0.5, -1.0, -2.0, 1.0, 0.4, 0.4,

  0.0,  1.0, -4.0, 0.4, 1.0, 0.4,  // Дальний зеленый треугольник
  -0.5, -1.0, -4.0, 0.4, 1.0, 0.4,
  0.5, -1.0, -4.0, 1.0, 0.4, 0.4,
]);
```

Если теперь запустить программу, вы увидите, что зеленый треугольник, который должен быть последним (дальним) оказался впереди (рис. 7.28).



Рис. 7.28. Зеленый треугольник, который должен быть последним (дальним) оказался впереди

Рисование объектов в заданном порядке, как это делает WebGL по умолчанию, может быть удачным решением, когда последовательность объектов можно определить заранее и в дальнейшем сцена не изменяется. Однако, при исследовании объекта с разных направлений, путем изменения положения точки наблюдения, невозможно заранее определить порядок следования объектов.

Удаление скрытых поверхностей

Для решения этой проблемы в WebGL имеется поддержка удаления скрытых поверхностей. Она удаляет поверхности, скрытые за объектами переднего плана, позволяя рисовать сцены так, чтобы объекты, находящиеся сзади, правильно скрывались за объектами, находящимися спереди, независимо от того, в каком порядке определяются вершины в тексте программы. Эта поддержка уже встроена в систему WebGL и ее просто нужно активизировать.

Чтобы активизировать удаление скрытых поверхностей в WebGL, требуется выполнить два следующих шага:

1. Активизировать механизм удаления скрытых поверхностей

```
gl.enable(gl.DEPTH_TEST);
```

2. Очистить буфер глубины, используемый этой функцией, перед рисованием.

```
gl.clear(gl.DEPTH_BUFFER_BIT);
```

Функция `gl.enable()`, используемая в шаге 1, в действительности активизирует несколько функций в WebGL.

`gl.enable(cap)`

Активизирует функцию, обозначенную аргументом `cap` (capability – функция).

Параметры	cap	Определяет активизируемую функцию
	<code>gl.DEPTH_TEST</code> ¹	Удаление скрытых поверхностей
	<code>gl.BLEND</code>	Смешивание (см. главу 9, «Иерархические объекты»)
	<code>gl.POLYGON_OFFSET_FILL</code>	Добавление сдвига к глубине (см. следующий раздел), и другие ²
Возвращаемое значение	нет	

¹ Выбор названия «DEPTH_TEST» для функции удаления скрытых поверхностей кажется немного странным, но в действительности это имя было выбрано, исходя из того факта, что система определяет, какие объекты рисовать на переднем плане, проверяя (TEST) глубину (DEPTH) местоположения каждого объекта.

² Хотя в этой книге не рассматриваются другие функции WebGL, которые можно активизировать с помощью `gl.enable()`, тем не менее мы перечислим их: `gl.CULL_FACE`, `gl.DITHER`, `gl.SAMPLE_ALPHA_TO_COVERAGE`, `gl.SAMPLE_COVERAGE`, `gl.SCISSOR_TEST` и `gl.STENCIL_TEST`. За дополнительной информацией обращайтесь к книге «OpenGL Programming Guide».

Буфер глубины (depth buffer), который очищает функция `gl.clear()` (шаг 2), – это внутренний буфер, используемый для удаления скрытых поверхностей. Рисование объектов и фигур осуществляется системой WebGL в буфере

цвета, содержимое которого затем отображается в элементе `<canvas>`, однако для удаления скрытых поверхностей требуется знать их глубину (расстояние от точки наблюдения). Эту информацию как раз и хранит буфер глубины (см. рис. 7.29). Направление в глубину совпадает с осью Z, поэтому иногда буфер глубины называют также z-буфером.



Рис. 7.29. Буфер глубины используется механизмом удаления скрытых поверхностей

Поскольку буфер глубины используется каждой командой рисования, его необходимо очищать перед каждой операцией рисования; в противном случае вы получите некорректные результаты. Чтобы очистить буфер глубины, функции `gl.clear()` необходимо передать `gl.DEPTH_BUFFER_BIT`:

```
gl.clear(gl.DEPTH_BUFFER_BIT);
```

До настоящего момента в своих программах мы очищали только буфер цвета. Так как теперь нам нужно очищать еще и буфер глубины, мы можем сделать это одним вызовом, выполнив поразрядную операцию «ИЛИ» (`|`) со значениями `gl.COLOR_BUFFER_BIT` (представляющим буфер цвета) и `gl.DEPTH_BUFFER_BIT` (представляющим буфер глубины), и передав результат в вызов `gl.clear()`:

```
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

Вы можете использовать поразрядную операцию «ИЛИ», как показано здесь, всякий раз, когда потребуется очистить сразу два буфера.

Чтобы деактивировать функцию, активированную с помощью `gl.enable()`, воспользуйтесь функцией `gl.disable()`.

`gl.disable(cap)`

Деактивирует функцию, обозначенную аргументом `cap` (capability – функция).

Параметры `cap` То же, что и в функции `gl.enable()`.

Возвращаемое значение нет

Ошибки `INVALID_ENUM` В аргументе `cap` передано недопустимое значение.

Пример программы (DepthBuffer.js)

Давайте добавим удаление скрытых поверхностей (шаги 1 и 2) в программу `PerspectiveView_mvMatrix.js` и сохраним ее под именем `DepthBuffer.js`. Обратите внимание, что порядок следования координат вершин в буферном объекте остался прежним, то есть треугольники будут рисоваться в направлении от ближних к дальним: сначала синие, затем желтые и, наконец, зеленые. Результат работы этой программы выйдет идентично результату, который производит программа `PerspectiveView_mvMatrix`. Исходный код программы приводится в листинге 7.10.

Листинг 7.10. `DepthBuffer.js`

```
1 // DepthBuffer.js
...
23 function main() {
...
41 var n = initVertexBuffers(gl);
...
47 // Определить цвет для очистки области рисования <canvas>
48 gl.clearColor(0, 0, 0, 1);
49 // Активировать поддержку удаления скрытых поверхностей
50 gl.enable(gl.DEPTH_TEST);
...
73 // Очистить буферы цвета и глубины
74 gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
75
76 gl.drawArrays(gl.TRIANGLES, 0, n); // Нарисовать треугольники
...
85 gl.drawArrays(gl.TRIANGLES, 0, n); // Нарисовать треугольники
86 }
87
88 function initVertexBuffers(gl) {
89 var verticesColors = new Float32Array([
90 // Координаты вершин и цвета
91 0.0, 1.0, 0.0, 0.4, 0.4, 1.0, // Ближний синий треугольник
92 -0.5, -1.0, 0.0, 0.4, 0.4, 1.0,
93 0.5, -1.0, 0.0, 1.0, 0.4, 0.4,
94
95 0.0, 1.0, -2.0, 1.0, 1.0, 0.4, // Желтый треугольник в середине
96 -0.5, -1.0, -2.0, 1.0, 1.0, 0.4,
97 0.5, -1.0, -2.0, 1.0, 0.4, 0.4,
98
99 0.0, 1.0, -4.0, 0.4, 1.0, 0.4, // Дальний зеленый треугольник
100 -0.5, -1.0, -4.0, 0.4, 1.0, 0.4,
101 0.5, -1.0, -4.0, 1.0, 0.4, 0.4,
102 ]);
103 var n = 9;
...
137 return n;
138 }
```


Если запустить программу `DepthBuffer`, можно увидеть, что она удаляет скрытые поверхности и объекты, расположенные сзади, правильно закрываются объектами, расположенными спереди. Этот пример наглядно показывает, что поддержка удаления скрытых поверхностей прекрасно справляется со своей задачей, независимо от местоположения точки наблюдения. Однако, этот пример также показывает, что для отображения практически любых трехмерных сцен, кроме самых простых, необходимо всегда активировать механизм удаления скрытых поверхностей и постоянно очищать буфер глубины перед каждой операцией рисования.

Обратите также внимание, что поддержка удаления скрытых поверхностей требует корректной настройки границ видимого объема. Если здесь допустить ошибку (использовать настройки WebGL по умолчанию), вы наверняка получите некорректные результаты. Видимый объем можно определить и в форме прямоугольного параллелепипеда, и в форме четырехгранной пирамиды.

Z-конфликт

Поддержка удаления скрытых поверхностей – сложная и мощная особенность системы WebGL, которая в большинстве действует безупречно. Однако иногда, когда две фигуры или два объекта располагаются слишком близко друг к другу, результат ее работы выглядит менее естественно. Этот феномен известен как **Z-конфликт** (Z-fighting) и показан на рис. 7.30. Здесь мы нарисовали два треугольника, имеющих одну и ту же координату Z.

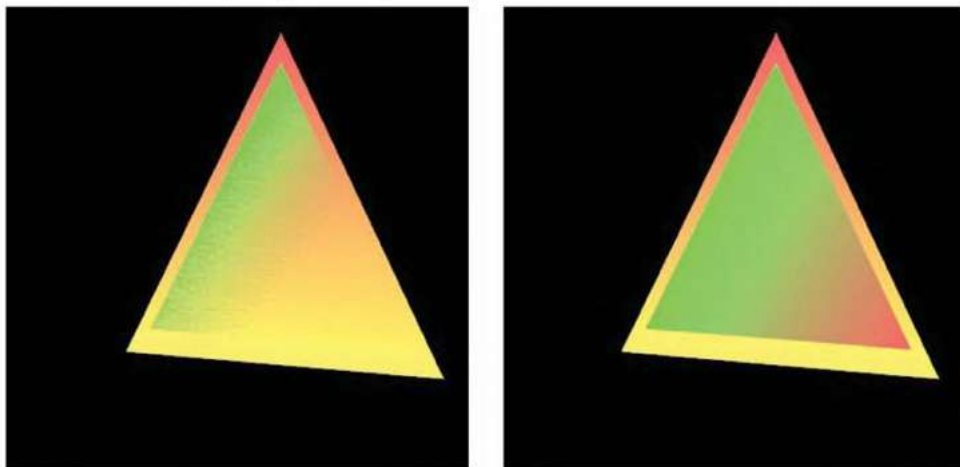


Рис. 7.30. Визуальный эффект, обусловленный Z-конфликтом (слева)

Z-конфликт возникает из-за того, что вычисления с буфером глубины имеют ограниченную точность и иногда система оказывается неспособна определить, какой объект находится спереди, а какой сзади. Технически этого конфликта можно избежать, если уделить особое внимание значениям координаты Z на этапе создания модели; однако, реализация такого решения выглядит нереалистичной в ситуациях, когда воспроизводится анимация с участием нескольких объектов.

Чтобы помочь решить эту проблему, WebGL предоставляет функцию, которая называется **добавление сдвига к глубине** (`polygon offset`). Эта функция автоматически выполняет сдвиг координаты Z, на величину, которая является функцией наклона объекта по отношению к линии взгляда. Достаточно добавить в программу всего две строки кода, чтобы активировать эту функцию.

1. Активировать функцию добавления сдвига к глубине вызовом:

```
gl.enable(gl.POLYGON_OFFSET_FILL);
```

2. Определить параметры, используемые при вычислении величины сдвига (перед рисованием):

```
gl.polygonOffset(1.0, 1.0);
```

Здесь `gl.enable()` – это тот же самый метод, с помощью которого мы активировали механизм удаления скрытых поверхностей, но на этот раз он должен вызываться с другим аргументом. Более подробное описание `gl.polygonOffset()` приводится ниже.

`gl.polygonOffset(factor, units)`

Определяет поправку к координате Z для каждой вершины, которая будет нарисована после вызова этого метода. Поправка вычисляется по формуле $m * factor + r * units$, где m представляет наклон треугольника относительно линии взгляда, а r – наименьшая разница между двумя значениями координаты Z, которые могут оказаться неразличимыми для WebGL.

Возвращаемое значение нет

Ошибки нет

Рассмотрим программу `zfighting`, использующую функцию добавления сдвига к глубине с целью уменьшить вероятность Z-конфликта (см. листинг 7.11).

Листинг 7.11. `Zfighting.js`

```
1 // Zfighting.js
...
23 function main() {
...
69 // Активировать функцию добавления сдвига к глубине
70 gl.enable(gl.POLYGON_OFFSET_FILL);
71 // Нарисовать треугольники
72 gl.drawArrays(gl.TRIANGLES, 0, n/2); // Зеленый треугольник
73 gl.polygonOffset(1.0, 1.0); // Установить параметры вычисления поправки
74 gl.drawArrays(gl.TRIANGLES, n/2, n/2); // Желтый треугольник
75 }
76
77 function initVertexBuffers(gl) {
78   var verticesColors = new Float32Array([
79     // Координаты вершин и цвета
80     0.0, 2.5, -5.0, 0.0, 1.0, 0.0, // Зеленый треугольник
81     -2.5, -2.5, -5.0, 0.0, 1.0, 0.0,
```

```
82     2.5, -2.5, -5.0, 1.0, 0.0, 0.0,  
83  
84     0.0, 3.0, -5.0, 1.0, 0.0, 0.0, // Желтый треугольник  
85     -3.0, -3.0, -5.0, 1.0, 1.0, 0.0,  
86     3.0, -3.0, -5.0, 1.0, 1.0, 0.0,  
87   });  
88   var n = 6;
```

Если внимательно рассмотреть исходный код программы, начиная со строки 80, можно увидеть, что все вершины имеют значение -5.0 координаты Z, из-за чего должен возникнуть Z-конфликт.

В оставшейся части кода, выполняется активация функции добавления сдвига (строка 70) и производится рисование зеленого и желтого треугольников (строки 72 и 74). Для простоты в этой программе используется единственный буферный объект, поэтому при вызове `gl.drawArrays()` требуется уделить особое внимание правильности значений второго и третьего аргументов. Второй аргумент представляет номер вершины, с которой должно начинаться рисование, а в третьем передается число вершин для рисования. После рисования зеленого треугольника программа определяет параметры для вычисления поправки вызовом метода `gl.polygonOffset()`. После этого ко всем вершинам, которые будут нарисованы, WebGL добавит смещение к координате Z. Если загрузить эту программу в браузер, можно увидеть два правильно нарисованных треугольника, без проявления эффектов, обусловленных Z-конфликтом, как показано на рис. 7.30 (справа). Если теперь закомментировать строку 73 и перезагрузить программу, можно увидеть эффект влияния Z-конфликта, как показано на рис. 7.30 слева.

Привет, куб

До настоящего момента мы рассказывали о различных возможностях WebGL, демонстрируя их на примере простых треугольников. Теперь вы обладаете достаточным объемом знаний, чтобы приступить к рисованию трехмерных объектов, и начнем мы с рисования куба, как показано на рис. 7.31. (Координаты всех вершин показаны на схеме справа.) Программа, исследованием которой мы займемся, называется `HelloCube`. В ней определяется восемь вершин, образующих куб, и их цвета: белый, пурпурный, красный, желтый, зеленый, светло-голубой, синий и черный. Как уже говорилось в главе 5, «Цвет и текстура», из-за того, что при растеризации происходит интерполяция цветов между вершинами, в результате получается куб, окрашенный плавными переходами цветов из одного в другой (фактически «цветовое тело», аналог 2-мерного «цветового круга»).

Рассмотрим случай, когда требуется нарисовать куб с применением уже известных вам команд, таких как `gl.drawArrays()`. В этой ситуации вам придется использовать один из следующих режимов рисования: `gl.TRIANGLES`, `gl.TRIANGLE_STRIP` или `gl.TRIANGLE_FAN`. Самый простой способ – рисование каждой грани, как состоящей из двух треугольников. Иными словами, грань с четырьмя вершинами (`v0`, `v1`, `v2`, `v3`) можно нарисовать, определив два набора

вершин для двух треугольников (v_0, v_1, v_2) и (v_0, v_2, v_3), и повторить ту же процедуру для всех остальных граней. В данном случае координаты вершин можно определить внутри буферного объекта, как показано ниже:

```
var vertices = new Float32Array([
    1.0, 1.0, 1.0, -1.0, 1.0, 1.0, -1.0, -1.0, 1.0, // v0, v1, v2
    1.0, 1.0, 1.0, -1.0, -1.0, 1.0, 1.0, -1.0, 1.0, // v0, v2, v3
    1.0, 1.0, 1.0, 1.0, -1.0, 1.0, 1.0, -1.0, -1.0, // v0, v3, v4
    ...
]);
```

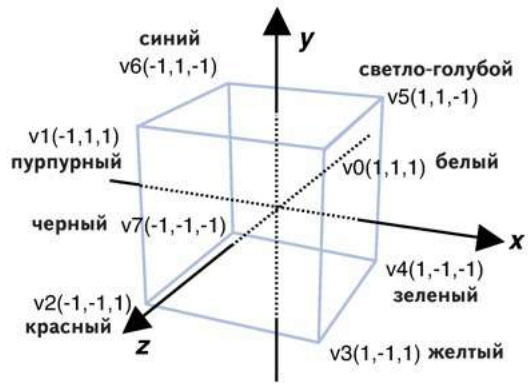


Рис. 7.31. HelloCube и координаты вершин куба

Так как одна грань состоит из двух треугольников, чтобы нарисовать ее требуется определить шесть вершин. Так как число граней равно 6, общее число вершин, которое потребуется определить, составляет $6 \times 6 = 36$. После определения координат для каждой из 36 вершин, их нужно сохранить в буферном объекте и затем вызвать метод `gl.drawArrays(gl.TRIANGLES, 0, 36)`, который нарисует куб. Такое решение требует определения 36 вершин, тогда как куб имеет всего 8, потому что все треугольники имеют общие вершины.

Однако есть возможность применить более экономный подход, рисуя каждую грань с использованием режима `gl.TRIANGLE_FAN`. Так как в режиме `gl.TRIANGLE_FAN` грань куба можно нарисовать, определив всего 4 вершины (v_0, v_1, v_2, v_3), в конечном итоге придется определить $4 \times 6 = 24$ вершины². Однако нам потребуется вызвать `gl.drawArrays()` отдельно для каждой грани (из шести). То есть, каждый из описанных подходов имеет свои достоинства и недостатки, но ни один из них не выглядит идеальным.

Как вы наверняка догадались, WebGL имеет решение этой проблемы: `gl.drawElements()`. Этот метод предоставляет альтернативный способ рисования трехмерных объектов в WebGL, требуя определить минимальное число вершин. Чтобы воспользоваться им, необходимо определить координаты вершин

² Число вершин можно сократить еще больше. При использовании режима `gl.TRIANGLE_STRIP` достаточно определить 14 вершин.

объекта и явно описать, какую фигуру (в данном случае – куб) должна нарисовать система WebGL.

Если разбить наш куб (рис. 7.31 справа) на вершины, образующие треугольники, мы получим структуру, изображенную на рис. 7.32. Взглянув на левую часть рисунка, можно заметить, что «Куб» указывает на список «Грани», название которого свидетельствует о том, что куб состоит из шести граней: передней, правой, левой, верхней, нижней и задней. Каждая грань, в свою очередь, состоит из двух треугольников, собранных в список «Треугольники». Числа в списке «Треугольники» представляют индексы в списке «Координаты». Нумерация индексов координат вершин начинается с нуля.

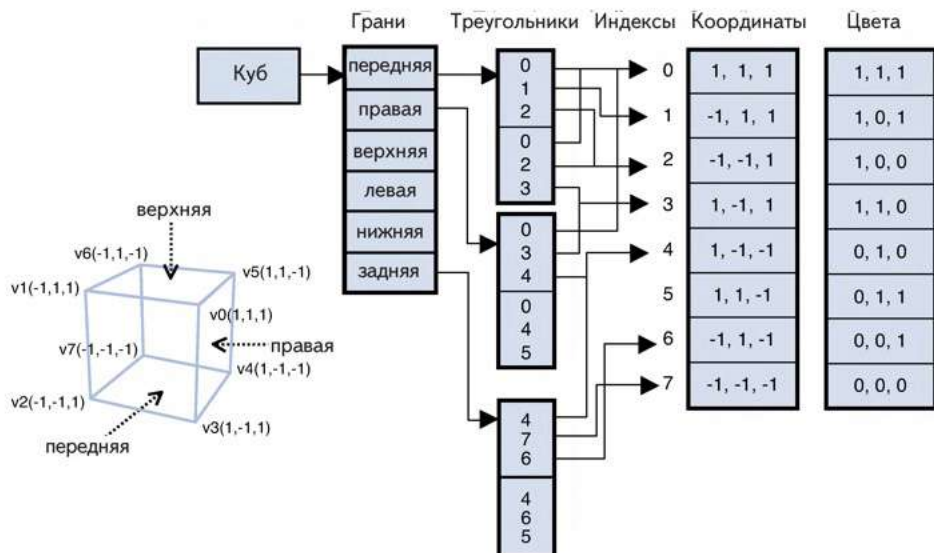


Рис. 7.32. Взаимосвязь между гранями, составляющими куб, треугольниками, вершинами и цветами

При таком подходе мы получаем структуру данных, описывающую, как построить объект (куб), исходя из координат вершин и цветов.

Рисование объектов с использованием индексов и координат вершин

До сих пор для рисования вершин мы пользовались методом `gl.drawArrays()`. Однако система WebGL поддерживает альтернативный подход, метод `gl.drawElements()`, напоминающий `gl.drawArrays()` и имеющий свои преимущества, с которыми мы познакомимся позднее. А пока посмотрим, как пользоваться методом `gl.drawElements()`. Для этого нужно определить буфер индексов типа `gl.ELEMENT_ARRAY_BUFFER`, а не `gl.ARRAY_BUFFER` (вводное описание можно найти в главе 3). Основное отличие буферов типа `gl.ELEMENT_ARRAY_BUFFER` состоит в том, что хранимые в них данные структурированы по индексам.

gl.drawElements(mode, count, type, offset)

Выполняет шейдер и рисует геометрическую фигуру в указанном режиме `mode`, используя индексы в буферном объекте типа `gl.ELEMENT_ARRAY_BUFFER`.

Параметры	<code>mode</code>	Определяет тип фигуры (см. рис. 3.17). Допустимыми значениями являются следующие константы: <code>gl.POINTS</code> , <code>gl.LINES</code> , <code>gl.LINE_STRIP</code> , <code>gl.LINE_LOOP</code> , <code>gl.TRIANGLES</code> , <code>gl.TRIANGLE_STRIP</code> и <code>gl.TRIANGLE_FAN</code> .
	<code>count</code>	Число индексов, участвующих в операции рисования (целое).
	<code>type</code>	Определяет тип индексов: <code>gl.UNSIGNED_BYTE</code> или <code>gl.UNSIGNED_SHORT</code> ¹
	<code>offset</code>	Определяет смещение в массиве индексов в байтах, откуда следует начать рисование.
Возвращаемое значение	нет	
Ошибки	<code>INVALID_ENUM</code>	В аргументе <code>mode</code> передано недопустимое значение.
	<code>INVALID_VALUE</code>	В аргументе <code>count</code> или <code>offset</code> передано отрицательное число.

¹ Даже если значение `type` не соответствует фактическому типу массива (`Uint8Array` или `Uint16Array`), указанному при создании буфера `gl.ELEMENT_ARRAY_BUFFER`, это не вызовет ошибку выполнения метода. Однако, если, к примеру, создать массив индексов типа `Uint16Array`, а данному методу передать в аргументе `type` значение `gl.UNSIGNED_BYTE`, это может привести к некорректному отображению фигуры.

Запись индексов в буферный объект, связанный с типом `gl.ELEMENT_ARRAY_BUFFER`, выполняется точно так же, как запись информации о вершинах в буферный объект, используемый с методом `gl.drawArrays()`. То есть, мы точно так же должны использовать методы `gl.bindBuffer()` и `gl.bufferData()`, с той лишь разницей, что в первом аргументе `target` им следует передать значение `gl.ELEMENT_ARRAY_BUFFER`. Рассмотрим пример программы.

Пример программы (HelloCube.js)

В листинге 7.12 приводится исходный код обсуждаемой программы. Вершинный и фрагментный шейдеры предполагают использование видимого объема в форме четырехгранной пирамиды и выполняют преобразование перспективной проекции, как в программе `ProjectiveView_mvpmatrix.js`. Важно понимать, что `gl.drawElements()` не делает ничего особенного. Вершинный шейдер просто преобразует координаты вершин, а фрагментный шейдер устанавливает цвета фрагментов, переписывая значение `varying`-переменной в `gl_FragColor`. Главное

отличие этой программы от всех предыдущих заключено в обработке буферного объекта в функции `initVertexBuffers()`.

Листинг 7.12. HelloCube.js

```
1 // HelloCube.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4     ...
5     'void main() {\n' +
6     '   gl_Position = u_MvpMatrix * a_Position;\n' +
7     '   v_Color = a_Color;\n' +
8     '}\n';
9
10 // Фрагментный шейдер
11 var FSHADER_SOURCE =
12     ...
13     'void main() {\n' +
14     '   gl_FragColor = v_Color;\n' +
15     '}\n';
16
17 function main() {
18     ...
19     // Определить координаты вершин и цвет
20     var n = initVertexBuffers(gl);
21     ...
22     // Определить цвет для очистки и включить удаление скрытых поверхностей
23     gl.clearColor(0.0, 0.0, 0.0, 1.0);
24     gl.enable(gl.DEPTH_TEST);
25     ...
26     // Определить положение точки наблюдения и форму видимого объема
27     var mvpMatrix = new Matrix4();
28     mvpMatrix.setPerspective(30, 1, 1, 100);
29     mvpMatrix.lookAt(3, 3, 7, 0, 0, 0, 0, 1, 0);
30
31     // Передать матрицу модели вида проекции matrix в u_MvpMatrix
32     gl.uniformMatrix4fv(u_MvpMatrix, false, mvpMatrix.elements);
33
34     // Очистить буферы цвета и глубины
35     gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
36
37     // Нарисовать куб
38     gl.drawElements(gl.TRIANGLES, n, gl.UNSIGNED_BYTE, 0);
39 }
40
41 function initVertexBuffers(gl) {
42     ...
43     var verticesColors = new Float32Array([
44         // Координаты вершин и цвета
45         1.0, 1.0, 1.0, 1.0, 1.0, 1.0, // v0 Белый
46         -1.0, 1.0, 1.0, 1.0, 0.0, 1.0, // v1 Пурпурный
47         -1.0, -1.0, 1.0, 1.0, 0.0, 0.0, // v2 Красный
48         ...
49         -1.0, -1.0, -1.0, 0.0, 0.0, 0.0 // v7 Черный
50     ]);
51 }
```

```

93
94 // Индексы вершин
95 var indices = new Uint8Array([
96     0, 1, 2, 0, 2, 3, // передняя
97     0, 3, 4, 0, 4, 5, // правая
98     0, 5, 6, 0, 6, 1, // верхняя
99     1, 6, 7, 1, 7, 2, // левая
100    7, 4, 3, 7, 3, 2, // нижняя
101    4, 7, 6, 4, 6, 5 // задняя
102 ]);
103
104 // Создать буферный объект
105 var vertexColorBuffer = gl.createBuffer();
106 var indexBuffer = gl.createBuffer();
107 ...
111 // Записать координаты и цвета в буферный объект
112 gl.bindBuffer(gl.ARRAY_BUFFER, vertexColorBuffer);
113 gl.bufferData(gl.ARRAY_BUFFER, verticesColors, gl.STATIC_DRAW);
114
115 var FSIZE = verticesColors.BYTES_PER_ELEMENT;
116 // Получить ссылку на переменную a_Position и разрешить присваивание ей
117 var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
118 ...
122 gl.vertexAttribPointer(a_Position, 3, gl.FLOAT, false, FSIZE * 6, 0);
123 gl.enableVertexAttribArray(a_Position);
124 // Получить ссылку на переменную a_Position и разрешить присваивание ей
125 var a_Color = gl.getAttribLocation(gl.program, 'a_Color');
126 ...
130 gl.vertexAttribPointer(a_Color, 3, gl.FLOAT, false, FSIZE * 6, FSIZE *
131 3);
132 gl.enableVertexAttribArray(a_Color);
133
134 // Записать индексы в буферный объект
135 gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
136 gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, indices, gl.STATIC_DRAW);
137
138 return indices.length;
139 }

```

Порядок выполнения операций в JavaScript-функции `main()` остался прежним, как в программе `ProjectiveView_mvpMatrix.js`, тем не менее коротко пройдемся по этой функции. После записи информации о вершинах в буферный объект вызовом `initVertexBuffers()` в строке 41, мы активируем удаление скрытых поверхностей (строка 49). Это необходимо, так как для корректного отображения куба требуется, чтобы система WebGL учитывала взаиморасположение граней.

В строках с 59 по 61 выполняется определение позиции точки наблюдения и формы видимого объема, и матрица модели вида проекции передается в вершинный шейдер через `uniform-переменную u_MvpMatrix`.

В строке 67 производится очистка буферов цвета и глубины и в строке 70 выполняется рисование куба вызовом `gl.drawElements()`. Использование метода `gl.drawElements()` является существенным отличием этой программы от `ProjectiveView_mvpMatrix.js`, поэтому расскажем об этой операции подробнее.

Запись координат вершин, цветов и индексов в буферный объект

Запись координат вершин и информации о цвете в переменную-атрибут в `initVertexBuffers()` выполняется точно так же. Но на этот раз, из-за того, что информация о вершинах не всегда будет использоваться в том порядке, в каком она хранится в буферном объекте, требуется дополнительно определить порядок их использования. Для этого порядок следования вершин задается в `verticesColors` в виде индексов. Проще говоря, информация о первой вершине в буферном объекте будет иметь индекс 0, информация о второй вершине – индекс 1, и так далее. Ниже приводится фрагмент программы с функцией `initVertexBuffers()`, где определяются индексы:

```
73 function initVertexBuffers(gl) {  
    ...  
82   var verticesColors = new Float32Array([  
83     // Координаты вершин и цвета  
84     1.0, 1.0, 1.0, 1.0, 1.0, 1.0, // v0 Белый  
85     -1.0, 1.0, 1.0, 1.0, 0.0, 1.0, // v1 Пурпурный  
    ...  
91     -1.0, -1.0, -1.0, 0.0, 0.0, 0.0 // v7 Черный  
92   ]);  
93  
94   // Индексы вершин  
95   var indices = new Uint8Array([  
96     0, 1, 2, 0, 2, 3, // передняя  
97     0, 3, 4, 0, 4, 5, // правая  
98     0, 5, 6, 0, 6, 1, // верхняя  
99     1, 6, 7, 1, 7, 2, // левая  
100    7, 4, 3, 7, 3, 2, // нижняя  
101    4, 7, 6, 4, 6, 5 // задняя  
102  ]);  
103  
104   // Создать буферный объект  
105   var vertexColorBuffer = gl.createBuffer();  
106   var indexBuffer = gl.createBuffer();  
    ...  
133   // Записать индексы в буферный объект  
134   gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);  
135   gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, indices, gl.STATIC_DRAW);  
136  
137   return indices.length;  
138 }
```

Как вы наверняка заметили, в строке 106 создается буферный объект (`indexBuffer`), куда записываются индексы. Эти индексы хранятся в массиве `indices`, который определяется в строке 95. Так как индексы являются целыми числами (0, 1, 2, ...), мы использовали массив целочисленного типа `Uint8Array` (массив беззнаковых 8-битных целых чисел). Если число индексов должно быть больше 256, следует использовать массив типа `Uint16Array`. Содержимым это-

го массива является список треугольников, как показано на рис. 7.33, где каждая группа из трех индексов ссылается на группу координат трех вершин. В общем случае массив индексов не требуется создавать вручную, потому что инструменты трехмерного моделирования, о которых рассказывается в следующей главе, обычно генерируют его автоматически, вместе с массивом информации о вершинах.

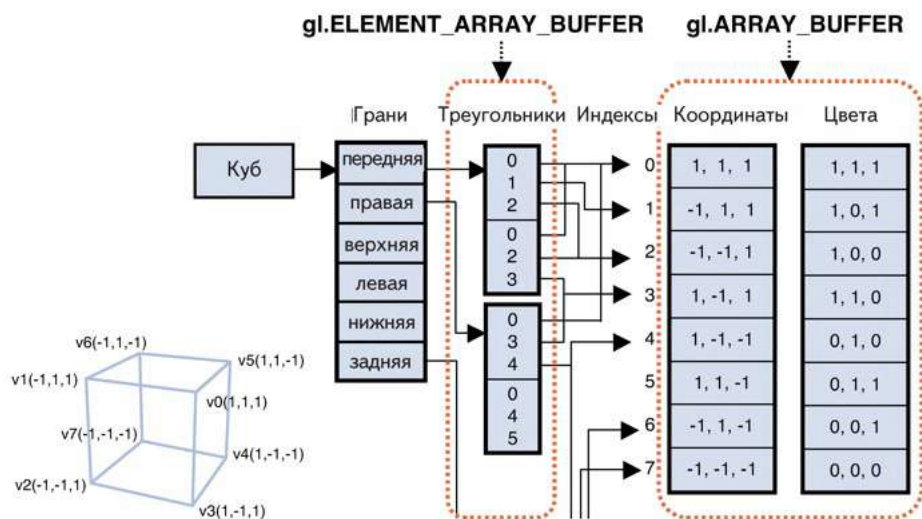


Рис. 7.33. Содержимое буферов типа `gl.ELEMENT_ARRAY_BUFFER` и `gl.ARRAY_BUFFER`

Установка индексов выполняется в строках 134 и 135. Эта операция напоминает операцию записи буферного объекта, которую мы использовали уже неоднократно, с той лишь разницей, что в первом аргументе теперь передается значение `gl.ELEMENT_ARRAY_BUFFER`, которое даст знать системе WebGL, что буфер хранит индексы.

На рис. 7.34 показано внутреннее состояние системы WebGL, которое она получит после выполнения функции `initVertexBuffers()`.

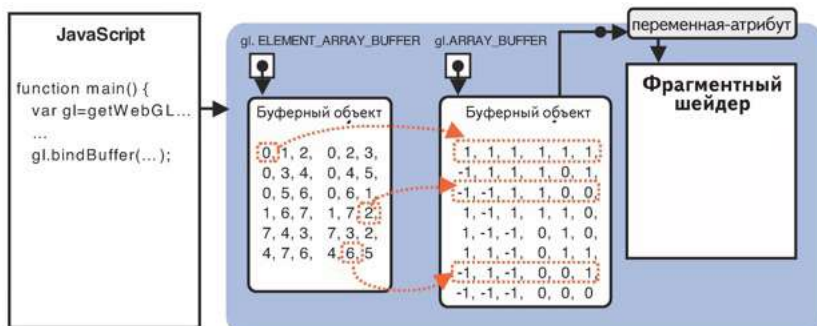


Рис. 7.34. `gl.ELEMENT_ARRAY_BUFFER` и `gl.ARRAY_BUFFER`

После выполнения всех подготовительных операций в строке 70 вызывается метод `gl.drawElements()`, чтобы нарисовать куб:

```
69 // Нарисовать куб
70 gl.drawElements(GL_TRIANGLES, n, GL_UNSIGNED_BYTE, 0);
```

Обратите внимание, что вторым аргументом в вызов `gl.drawElements()` передается число индексов, представляющих координаты вершин, которое не равно числу вершин, хранящихся в буферном объекте типа `gl.ARRAY_BUFFER`.

Метод `gl.drawElements()` извлечет индексы из буферного объекта `indexBuffer`, связанного с типом `gl.ELEMENT_ARRAY_BUFFER`, а информацию о вершинах – из буферного объекта `vertexColorBuffer`, связанного с типом `gl.ARRAY_BUFFER`. Затем информация будет передана в переменную-атрибут. Эта процедура повторится для каждого индекса, после чего будет выполнено рисование куба. При таком подходе есть возможность многократно использовать одну и ту же информацию о вершинах, благодаря тому, что она извлекается с помощью индексов. Несмотря на то, что `gl.drawElements()` позволяет экономить память за счет многократного использования информации о вершинах, процедура рисования требует дополнительных вычислительных ресурсов для преобразования индексов в информацию о вершинах. Это означает, что выбор между `gl.drawElements()` и `gl.drawArrays()`, имеющих свои достоинства и недостатки, в конечном счете будет зависеть от конкретной реализации системы.

На данном этапе совершенно очевидно, что метод `gl.drawElements()` обеспечивает весьма эффективный способ рисования трехмерных фигур. Но мы не рассмотрели еще одну важную особенность – возможность управления цветом, чтобы можно было нарисовать куб, окрашивая грани сплошным цветом.

Например, представьте, что нам потребовалось нарисовать куб, как показано на рис. 7.35, или наложить текстуру на его грани. Но, даже если мы будем заранее иметь информацию о цвете или о текстуре для каждой грани, мы не сможем реализовать задуманное с применением изображенной на рис. 7.33 комбинации индексов, списка треугольников и координат вершин.

В следующем примере мы посмотрим, как решить эту проблему и как определить цвет для каждой грани.

Добавление цвета для каждой грани куба

Как уже говорилось выше, в вершинный шейдер можно передать только информацию о вершинах. Соответственно, нам нужен некоторый способ, который позволил бы передать в вершинный шейдер не только информацию о вершинах треугольников,

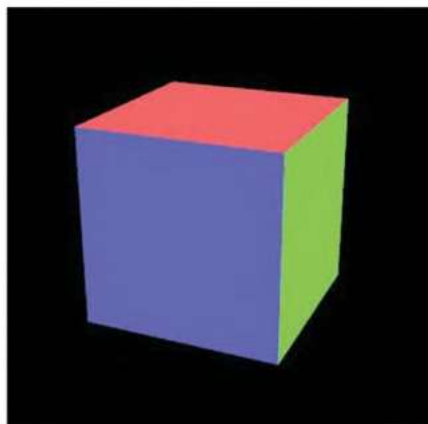


Рис. 7.35. Куб с гранями разного цвета

но и цвет грани. Например, чтобы нарисовать переднюю грань синим цветом, состоящую из вершин v_0 , v_1 , v_2 и v_3 (рис. 7.33), необходимо для каждой вершины определить синий цвет.

Однако, как вы наверняка заметили, вершина v_0 также входит в состав правой и верхней граней. Поэтому, если для вершин передней грани указать синий цвет, мы не сможем выбрать иной цвет для тех же вершин, принадлежащих другим граням. Чтобы справиться с этой проблемой, можно создать дублирующие записи для всех общих вершин в списке координат (хотя это и неэффективное решение), как показано на рис. 7.36. При этом нам придется обрабатывать общие вершины с идентичными координатами, как отдельные сущности.³

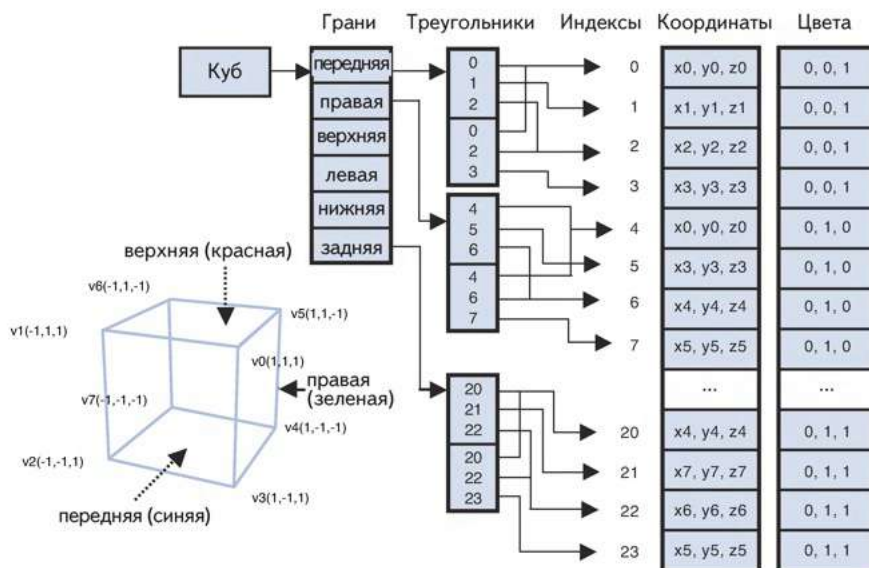


Рис. 7.36. Взаимосвязь между гранями, составляющими куб, треугольниками и вершинами (определенными так, что для каждой грани можно назначать свой цвет)

В такой конфигурации содержимое списка индексов, состоящего из списка треугольников граней, отличается от списка граней, что дает возможность определять цвет для каждой грани независимо. Этот же подход можно использовать для наложения текстур на грани. Можно было бы определить координаты текстуры для каждой вершины, но, фактически, то же самое можно сделать, заменив значения цветов в списке (рис. 7.36) координатами текстуры. Подробнее этот подход рас-

³ Если разбить все грани на треугольники и рисовать их с помощью `gl.drawArrays()`, придется определить $6 \text{ вершин} * 6 \text{ граней} = 36 \text{ вершин}$, что лишит нас выигрыша в памяти, который дает `gl.drawElements()` в сравнении с `gl.drawArrays()`. Это обусловлено тем, что куб, или кубоид, является особым трехмерным объектом, в каждой вершине которого сходятся три грани, из-за чего для каждой вершины требуется определить три цвета. Однако в более сложных трехмерных моделях потребность определения нескольких цветов для одной вершины возникает довольно редко.

сматривается в разделе «Вращение объекта», в главе 10.

А теперь рассмотрим пример программы `ColoredCube`, отображающей куб, каждая грань которого окрашена в собственный цвет. Результат работы программы `ColoredCube` напоминает изображение на рис. 7.35.

Пример программы (*ColoredCube.js*)

Исходный код программы приводится в листинге 7.13. Поскольку единственное отличие от `HelloCube.js` заключается в способе хранения информации о вершинах в буферном объекте, более детально исследуем код, имеющий отношение к функции `initVertexBuffers()`. Ниже перечислены основные отличия от `HelloCube.js`:

- в `HelloCube.js` координаты вершин и цвета хранятся в едином буферном объекте, но так как из-за этого массив получается слишком громоздким, в данной программе координаты и цвета хранятся в разных буферных объектах;
- содержимое массива вершин (где хранятся координаты вершин), цвета (где хранится информация о цвете) и индексов (где хранятся индексы) изменено в соответствии с конфигурацией, представленной на рис. 7.36 (строки 83, 92 и 101);
- чтобы сохранить программу максимально компактной, определена функция `initArrayBuffer()`, которая выполняет создание буферного объекта, его связывание и запись данных в него (строки 116, 119 и 129).

В процессе исследования программы обратите внимание, как реализован второй пункт списка.

Листинг 7.13. `ColoredCube.js`

```
1 // ColoredCube.js
...
23 function main() {
    ...
40 // Определить информацию о вершинах
41 var n = initVertexBuffers(gl);
    ...
69 // Нарисовать куб
70 gl.drawElements(gl.TRIANGLES, n, gl.UNSIGNED_BYTE, 0);
71 }
72
73 function initVertexBuffers(gl) {
    ...
83 var vertices = new Float32Array([ // Координаты вершин
84     1.0, 1.0, 1.0, -1.0, 1.0, 1.0, -1.0, -1.0, 1.0, 1.0, -1.0, 1.0,
85     1.0, 1.0, 1.0, 1.0, -1.0, 1.0, 1.0, -1.0, -1.0, 1.0, 1.0, -1.0,
86     1.0, 1.0, 1.0, 1.0, 1.0, -1.0, -1.0, 1.0, -1.0, -1.0, 1.0, 1.0,
    ...
89     1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, 1.0, -1.0, 1.0, 1.0, -1.0
90 ]);
91 }
```

```

92  var colors = new Float32Array([ // Цвета
93      0.4, 0.4, 1.0,    0.4, 0.4, 1.0,    0.4, 0.4, 1.0,    0.4, 0.4, 1.0,
94      0.4, 1.0, 0.4,    0.4, 1.0, 0.4,    0.4, 1.0, 0.4,    0.4, 1.0, 0.4,
95      1.0, 0.4, 0.4,    1.0, 0.4, 0.4,    1.0, 0.4, 0.4,    1.0, 0.4, 0.4,
96      ...
97      0.4, 1.0, 1.0,    0.4, 1.0, 1.0,    0.4, 1.0, 1.0,    0.4, 1.0, 1.0
98  ]);
99
100
101  var indices = new Uint8Array([ // Индексы вершин
102      0, 1, 2, 0, 2, 3, // передняя грань
103      4, 5, 6, 4, 6, 7, // правая
104      8, 9, 10, 8, 10, 11, // верхняя
105      ...
106      20, 21, 22, 20, 22, 23 // задняя
107  ]);
108
109
110  // Создать буферный объект
111  var indexBuffer = gl.createBuffer();
112  ...
113  // Записать координаты вершин и цвета в буферный объект
114  if (!initArrayBuffer(gl, vertices, 3, gl.FLOAT, 'a_Position'))
115      return -1;
116
117  if (!initArrayBuffer(gl, colors, 3, gl.FLOAT, 'a_Color'))
118      return -1;
119  ...
120  // Записать индексы в буферный объект
121  gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
122  gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, indices, gl.STATIC_DRAW);
123
124  return indices.length;
125 }
126
127
128
129 function initArrayBuffer(gl, data, num, type, attribute) {
130     var buffer = gl.createBuffer(); // Создать буферный объект
131     ...
132     // Записать данные в буферный объект
133     gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
134     gl.bufferData(gl.ARRAY_BUFFER, data, gl.STATIC_DRAW);
135     // Присвоить буферный объект переменной-атрибуту
136     var a_attribute = gl.getAttribLocation (gl.program, attribute);
137     ...
138     gl.vertexAttribPointer(a_attribute, num, type, false, 0, 0);
139     // Разрешить присваивание буферного объекта переменной-атрибуту
140     gl.enableVertexAttribArray(a_attribute);
141
142     return true;
143 }
144
145 }

```

Эксперименты с примером программы

В программе `ColoredCube` мы определили свой цвет для каждой грани. Но, что случится, если мы определим один цвет для всех граней? Например, давайте попробуем в массиве `colors` указать белый цвет во всех его элементах, как показано

ниже, и назовем измененную программу `ColoredCube_singleColor.js`:

```
1 // ColoredCube_singleColor.js
...
92 var colors = new Float32Array([
93   1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
94   1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
...
98   1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0
99 ]);
```

После запуска этой программы на экране появится белый куб, как показано на рис. 7.37. Как видите, при использовании одного цвета оказывается труднее опознать куб в полученном изображении. До настоящего момента мы легко могли различать разные грани, благодаря разной их окраске, и легко опознавать фигуру, которую они составляют. Однако, после перехода на единый цвет, потерялось ощущение объема.

В реальном мире такого не происходит. Если положить на стол белый кубик, мы без труда опознаем его как кубик (см. рис. 7.38). Это объясняется тем, что каждая грань, несмотря на то, что все они одного цвета, выглядит несколько иначе, благодаря игре теней и света. В программе `ColoredCube_singleColor` такой эффект не запрограммирован, поэтому ребра и грани куба сливаются, усложняя распознавание формы фигуры. В следующей главе мы расскажем, как правильно настроить освещение в трехмерных сценах.

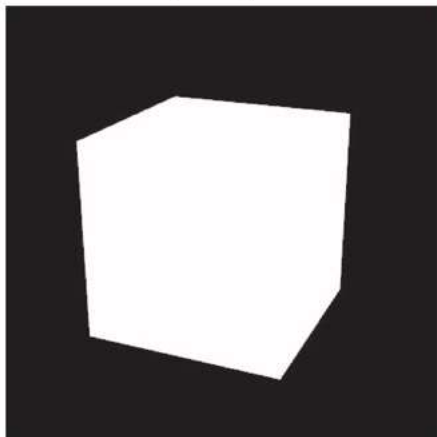


Рис. 7.37. Куб с гранями одного цвета



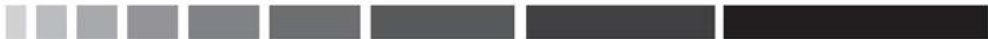
Рис. 7.38. Как выглядит белый кубик в реальном мире

В заключение

В этой главе мы исследовали приемы создания эффекта глубины через определение позиции точки наблюдения и формы видимого объема. Узнали, как рисовать настоящие трехмерные объекты, и коротко познакомились с локальной и мировой

системами координат. Многие примеры, рассматривавшиеся здесь, схожи с примерами из предыдущих глав, где демонстрировались приемы рисования 2-мерных фигур. Единственное существенное их отличие заключается в введении в дело оси Z для создания эффекта глубины.

В следующей главе рассказывается, как настроить освещение трехмерной сцены, а также как рисовать трехмерные фигуры и управлять ими, используя комплексные структуры. Мы также вернемся к функции `initShaders()`, решающей множество сложных проблем, с которыми вы теперь готовы встретиться.



ГЛАВА 8.

Освещение объектов

Основное внимание в этой главе уделяется освещению объектов, исследованию разных источников освещения и их влияния на трехмерные сцены. Освещение играет важную роль в придании реалистичности трехмерным сценам, потому что правильно подобранное освещение придает ощущение глубины.

В этой главе обсуждаются следующие ключевые моменты:

- тени, затенение и разнотипные источники света, включая точечные, направленные и рассеянные;
- отражение света в трехмерных сценах и два основных типа отражения: рассеянное и фоновое;
- тонкости затенения и как реализовать эффект освещения, чтобы объекты выглядели трехмерными.

К концу этой главы вы получите все знания, необходимые для создания эффекта освещения трехмерных сцен, наполненных простыми и сложными трехмерными объектами.

Освещение трехмерных объектов

Когда в реальном мире свет падает на объект, часть света отражается поверхностью этого объекта. И только когда отраженный свет попадает на сетчатки наших глаз, мы можем видеть объекты и различать их цвет. Например, белый кубик отражает белый свет. Этот свет попадает на сетчатку глаза и мы можем сказать, что кубик имеет белый цвет.

В реальном мире наблюдаются два важных явления, когда свет падает на объект (см. рис. 8.1):

- в зависимости от положения и направленности источника света цвет поверхности затеняется;
- в зависимости от положения и направленности источника света объекты «отбрасывают» тень.

В реальном мире мы обычно легко замечаем тень, но очень часто не обращаем внимание на затенение, которое придет объем трехмерным объектам. Затенение является не самым ярко выраженным эффектом, но оно присутствует всегда. Как

показано на рис. 8.1, даже грани чисто белого кубика несколько отличаются цветом, потому что каждая грань затенена по-разному. Как можно видеть на этом рисунке, поверхности, на которые падает больше света, выглядят ярче, а поверхности, на которые падает меньше света, выглядят темнее. Эти различия помогают нам различать грани и уверенно говорить, что мы видим именно кубик.

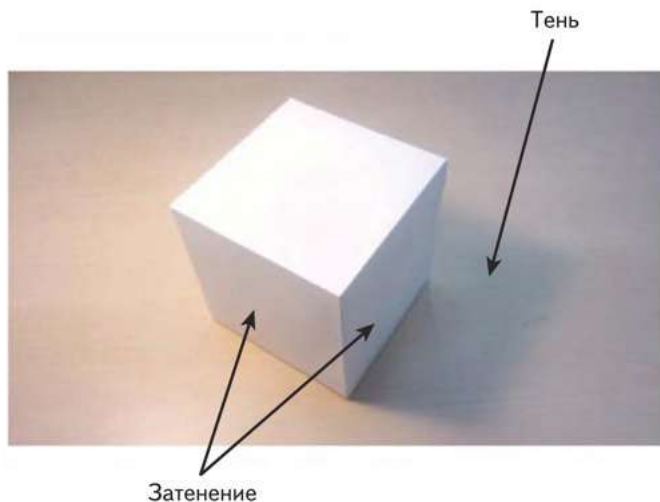


Рис. 8.1. Тень и затенение

В трехмерной графике термин **затенение** (shading)¹ используется для описания процедуры придания граням различной окраски, обусловленной разностью их освещенности. Другим эффектом является **тень**, когда объект отбрасывает тень на другую поверхность. В этом разделе мы обсудим приемы создания эффекта затенения, а воссоздание эффекта отбрасывания тени будет обсуждаться в главе 10, описывающей множество интересных приемов, основанных на базовых особенностях WebGL.

При обсуждении эффекта затенения следует учитывать две вещи:

- тип источника света;
- как свет отражается от поверхностей объекта;

Прежде чем перейти к программному коду, познакомимся с типами источников света и как свет отражается разными поверхностями.

Типы источников света

Объекты освещаются светом, испускаемым **источниками света**. В реальном мире источники света делятся на две основные категории: **источники направленного**

¹ Затенение играет в трехмерной графике настолько важную роль, что даже язык шейдеров GLSL ES был назван «OpenGL ES Shading Language» (дословно название «shading language» переводится, как «язык обработки теней», или «язык обработки полутонов» – Прим. перев.). Первоначально шейдеры предназначались для воссоздания эффекта затенения (shading).

света, такие как солнце, дающее естественное освещение, и **источники точечного освещения**, такие как лампы, дающие искусственное освещение. Кроме того, существуют еще **источники рассеянного освещения**, такие как стены и другие объекты, испускающие отраженный свет (см. рис. 8.2). В трехмерной графике существуют также дополнительные источники света. Например, световые блики, получаемые в результате отражения света фар, прожекторов и так далее. Однако в этой книге мы не будем рассматривать дополнительные источники света. А желающие могут обратиться за дополнительной информацией к книге «OpenGL ES 2.0 Programming Guide».



Рис. 8.2. Направленный свет, точечный свет и рассеянный свет

В этой книге все внимание будет уделяться трем основным типам источников света:

- **Источник направленного света:** источником направленного света называется такой источник, лучи от которого распространяются параллельно друг другу. Считается, что такой источник света находится чрезвычайно далеко, как Солнце от Земли. Из-за большого расстояния, когда лучи света достигают поверхности объекта, они оказываются практически параллельными. Такой источник света является самым простым, так как характеризуется только направлением и цветом, из-за того, что лучи от него распространяются параллельно друг другу.
- **Точечный источник света:** точечным источником света называется такой источник, который испускает лучи в разные стороны из единственной точки. Примером точечного источника света может служить лампа, фонарь, факел и так далее. Источники света этого типа характеризуются местоположением и цветом.² Направление распространения света от точечного источника определяется его местоположением. В силу этого в разных участках сцены свет может иметь разное направление распространения.

² Источники света этого типа дают затухающее освещение, то есть, сила света падает с увеличением расстояния от источника. Однако в этой книге, для простоты обсуждения и устранения ненужных сложностей из программного кода, мы будем полагать, что свет от таких источников не затухает. Если вам интересен этот аспект точечных источников света, обращайтесь к книге «OpenGL ES 2.0 Programming Guide».

- **Источник рассеянного освещения:** примерами источников рассеянного освещения могут служить другие источники света (направленные или точечные) и другие объекты, такие как стены, испускающие отраженный свет. Рассеянный свет освещает объекты со всех сторон и имеет одинаковую интенсивность во всех точках сцены.³ Например, если ночью открыть дверь холодильника, во всей кухне станет немного светлее. Это и есть эффект освещения рассеянным светом. Источник рассеянного света не имеет определенного местоположения и характеризуется только цветом.

Теперь, после знакомства с разными типами источников света, посмотрим, как свет отражается от поверхности объекта.

Типы отраженного света

Как свет отражается от поверхности объекта, и какой цвет приобретает отражающая поверхность, определяется двумя факторами: типом освещения и типом отражающей поверхности. Информация о типе освещения включает цвет и направление распространения света. Информация о поверхности включает ее цвет и ориентацию.

Свет, отраженный от поверхностей, делится на две основные категории: **диффузный** (или **рассеянный**) и **фоновый** (или **естественный**). В оставшейся части этого раздела описывается, как определить цвет отраженного света на основе двух параметров, описанных выше. При этом мы затронем математический аппарат, лежащий в основе этих вычислений, но он не слишком сложен.

Модель диффузного отражения

Диффузный отраженный свет – это результат отражения света от направленного или точечного источника света. Диффузный отраженный свет распространяется (рассеивается) во всех направлениях (см. рис. 8.3). Если поверхность совершенно гладкая, как зеркало, она отражает весь падающий на нее свет; однако, большинство поверхностей имеют шероховатости, как бумага, камни или пластик. В таких случаях свет рассеивается в случайных направлениях, отражаясь от мелких неровностей. Это явление представляет модель диффузного отражения.

В модели диффузного отражения цвет поверхности определяется цветом и направлением на источник света, а также основным цветом самой поверхности и ее ориентацией. Угол между направлением на источник света и ориентацией поверхности определяется как угол, образуемый направлением на источник света и направлением «перпендикуляра» к поверхности. Назовем этот угол θ (тэта). Тогда цвет поверхности в модели диффузного отражения можно вычислить с помощью формулы 8.1.

³ Фактически, рассеянный свет является суммой света, испускаемого многочисленными источниками, и отраженного различными поверхностями. Освещенность рассеянным светом учитывается лишь приближенно, потому что иначе потребовалось бы производить весьма сложные вычисления, чтобы учесть все имеющиеся источники и все направления, в которых распространяется отраженный свет.

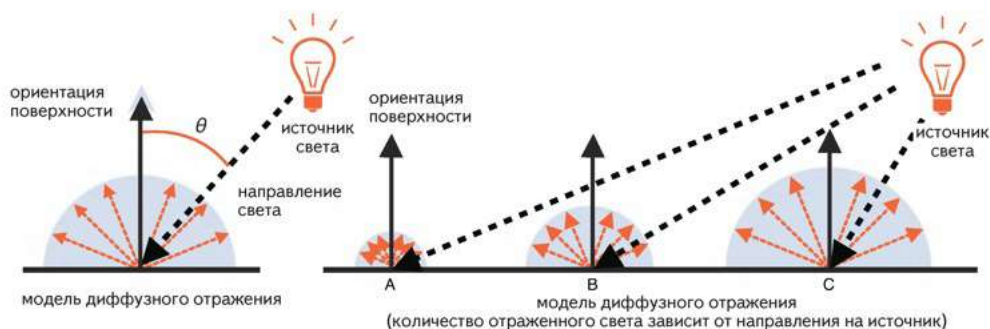


Рис. 8.3. Модель диффузного отражения

Формула 8.1.

$$\langle \text{цвет поверхности в модели диффузного отражения} \rangle = \langle \text{цвет света} \rangle \times \langle \text{собственный цвет поверхности} \rangle \times \cos \theta$$

где $\langle \text{цвет света} \rangle$ – это цвет света, испускаемого направленным или точечным источником света. Умножение на $\langle \text{собственный цвет поверхности} \rangle$ выполняется для каждого RGB-компонента цвета. Поскольку в модели диффузного отражения свет рассеивается во всех направлениях одинаково, интенсивность света, отраженного от определенной точки, не зависит от угла наблюдения (см. рис. 8.4).

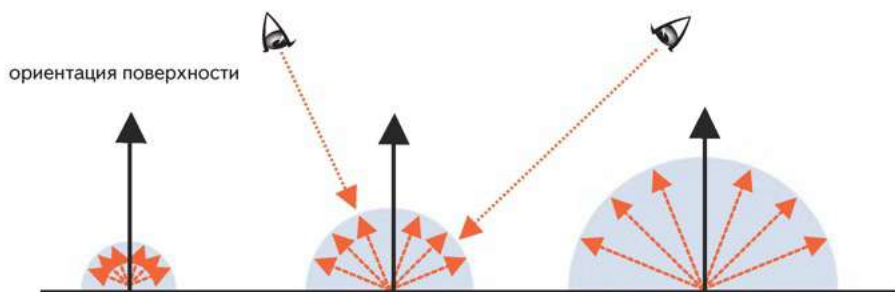


Рис. 8.4. Интенсивность света, отраженного от определенной точки, не зависит от угла наблюдения

Модель фонового отражения

Фоновый отраженный свет – это результат отражения света от другого источника света. В модели фонового отражения свет отражается под тем же углом, под каким он падает на поверхность. Так как фоновый свет освещает объект со всех сторон равномерно, яркость отраженного света не зависит от точки наблюдения (см. рис. 8.5). Цвет поверхности в модели фонового отражения можно вычислить с помощью формулы 8.2.

Формула 8.2.

$$\langle \text{цвет поверхности в модели фонового отражения} \rangle = \langle \text{цвет света} \rangle \times \langle \text{собственный цвет поверхности} \rangle$$



Рис. 8.5. Фоновое отражение

Когда применяются обе модели отражения, диффузного и фонового, цвет поверхности определяется как их сумма.

Формула 8.3.

$$\begin{aligned} \langle \text{цвет поверхности при диффузном и фоновом отражении} \rangle = \\ \langle \text{цвет в модели диффузного отражения} \rangle + \\ \langle \text{цвет в модели фонового отражения} \rangle \end{aligned}$$

Обратите внимание, что не всегда требуется использовать обе модели отражения или применять формулы в точности, как они указаны здесь. Вы свободно можете изменить любую из формул для достижения требуемого эффекта.

А теперь напишем несколько программ, реализующих затенение (затенение и раскраску поверхностей объекта, поместив источник света в определенное местоположение). Сначала попробуем реализовать затенение при направленном освещении в модели диффузного отражения.

Затенение при направленном освещении в модели диффузного отражения

Как описывалось в предыдущем разделе, в модели диффузного отражения цвет поверхности определяется направлением на источник света и ориентацией поверхности. Вычисление цвета при направленном освещении осуществляется просто, потому что направление лучей остается постоянным. Ниже еще раз приводится формула (формула 8.1) определения цвета поверхности в модели диффузного отражения:

$$\begin{aligned} \langle \text{цвет поверхности в модели диффузного отражения} \rangle = \\ \langle \text{цвет света} \rangle \times \langle \text{собственный цвет поверхности} \rangle \times \cos \theta \end{aligned}$$

Для вычислений нам потребуется следующая информация:

- цвет света, испускаемого источником (цвет направленного света);
- собственный цвет поверхности;
- угол (θ) между направлением на источник света и нормалью к поверхности.

Цвет испускаемого света может быть белым, как, например, солнечный свет, или любым другим, например оранжевым, как от ламп в дорожном туннеле. Как вы уже знаете, цвет может быть представлен тремя составляющими RGB. Белый

свет имеет RGB-значение (1.0, 1.0, 1.0). Под собственным цветом поверхности подразумевается цвет, который первоначально был определен для этой поверхности, например, красный или синий. Чтобы вычислить цвет поверхности, необходимо применить формулу к каждому из трех RGB-компонентов; то есть, вычисления производятся трижды.

Например, предположим, что свет, испускаемый источником, имеет белый цвет (1.0, 1.0, 1.0), а сама поверхность имеет красный цвет (1.0, 0.0, 0.0). Исходя из формулы 8.1, для угла θ равного 0.0 (то есть, когда свет падает перпендикулярно), $\cos \theta$ равен 1.0. Так как компонент R испускаемого света равен 1.0, компонент R собственного цвета поверхности равен 1.0 и $\cos \theta$ равен 1.0, компонент R отраженного света в модели диффузного отражения будет равен:

$$R = 1.0 * 1.0 * 1.0 = 1.0$$

Компоненты G и B вычисляются аналогично:

$$G = 1.0 * 0.0 * 1.0 = 0.0$$

$$B = 1.0 * 0.0 * 1.0 = 0.0$$

Исходя из этих вычислений, когда белый свет падает перпендикулярно на красную поверхность, цвет поверхности в модели диффузного отражения превращается в красный (1.0, 0.0, 0.0). Это в точности соответствует тому, что мы наблюдаем в реальности. Аналогично, когда красный свет падает на белую поверхность, результат получается тот же самый.

Теперь рассмотрим случай, когда угол θ равен 90 градусам, то есть, когда свет вообще не попадает на поверхность. Из своего опыта мы знаем, что в этом случае поверхность будет выглядеть черной. Проверим это. Так как $\cos \theta$ равен 0 (для угла θ , равного 90 градусам), в формулах, представленных выше, происходит умножение на 0, в результате компоненты R, G и B получают значение 0; то есть, цвет поверхности становится черным (0.0, 0.0, 0.0), как и ожидалось. Аналогично, когда угол θ равен 60 градусам, вполне ожидаемо, что какая-то доля света упадет на красную поверхность и она приобретет темно-красный цвет: так как $\cos \theta$ равен 0.5, цвет поверхности будет определен, как (0.5, 0.0, 0.0), то есть темно-красный, что, собственно и ожидалось.

Эти простые примеры наглядно показывают, как определяется цвет поверхности в модели диффузного отражения. Давайте упростим предыдущую формулу, чтобы с ней было проще работать и мы смогли бы исследовать рисование куба, освещенного направленным светом.

Использование направления света и ориентации поверхности в модели диффузного отражения

В предыдущих примерах мы выбирали произвольные значения угла θ . Однако на практике бывает довольно сложно получить угол θ между направлением на источник света и ориентацией поверхности. Например, при создании модели невозможно заранее определить, под каким углом будет падать свет на каждую поверх-

ность. Напротив, ориентация каждой поверхности не зависит от направлением на источник света. Так как направление света определяется местоположением его источника, кажется, имело бы смысл использовать в формуле эти две характеристики.

К счастью, математика говорит, что $\cos \theta$ можно получить, как скалярное произведение направления на источник света и ориентации поверхности. Так как в компьютерной графике скалярное произведение требуется вычислять достаточно часто, в GLSL ES была добавлена специальная функция.⁴ (Подробности можно найти в приложении В, «Встроенные функции в языке GLSL ES 1.0».) Скалярное произведение (которое в этой книге будет обозначаться символом « $\cdot$ »), представляющее значение $\cos \theta$, вычисляется следующим образом:

$$\cos \theta = \langle \text{направление света} \rangle \cdot \langle \text{ориентация поверхности} \rangle$$

Отсюда формулу 8.1 можно преобразовать в формулу 8.4:

Формула 8.4.

$$\begin{aligned} \langle \text{цвет поверхности в модели диффузного отражения} \rangle = \\ \langle \text{цвет света} \rangle \times \langle \text{собственный цвет поверхности} \rangle \times \\ \langle \text{направление света} \rangle \cdot \langle \text{ориентация поверхности} \rangle \end{aligned}$$

Остановимся на двух важных значениях: длина вектора и направление на источник света. Во-первых, длины векторов, представляющих направление на источник света и ориентацию поверхности, таких как (2.0, 2.0, 1.0), должны быть равны 1.0,⁵ иначе цвет поверхности будет получаться темнее или светлее. Корректировка компонентов вектора, чтобы сделать его длину равной 1.0, называется **нормализацией**.⁶ В языке GLSL ES имеется готовая функция нормализации векторов, которую можно использовать непосредственно.

Второй важный момент – направление распространения отраженного света. Отраженный свет распространяется в направлении, противоположном падению света, испускаемого источником (см. рис. 8.6).

Поскольку мы не используем углы для определения ориентации поверхностей, нам необходим какой-то другой механизм для этого. Решение заключается в использовании векторов нормалей.

⁴ Математически, скалярное произведение двух векторов, n и l , записывается, как:

$$n \cdot l = |n| \times |l| \times \cos \theta$$

где вертикальные прямые скобки ($| \ |$) означают длину вектора. Из этой формулы видно, что когда длины векторов n и l равны 1.0, скалярное произведение равно $\cos \theta$. Если предположить, что вектор n определяется как (n_x, n_y, n_z) , а вектор l – как (l_x, l_y, l_z) , тогда по закону косинусов:

$$n_l = n_x * l_x + n_y * l_y + n_z * l_z$$

⁵ Если предположить, что вектор n определяется как (n_x, n_y, n_z) , его длина вычисляется следующим образом: длина $n = |n| = \sqrt{n_x^2 + n_y^2 + n_z^2}$.

⁶ Нормализованный вектор n определяется как $(n_x/m, n_y/m, n_z/m)$, где m – длина вектора n . $|n| = \text{sqrt}(9) = 3$. Вектор (2.0, 2.0, 1.0), упомянутый выше, в нормализованном виде будет представлен как (2.0/3.0, 2.0/3.0, 1.0/3.0).

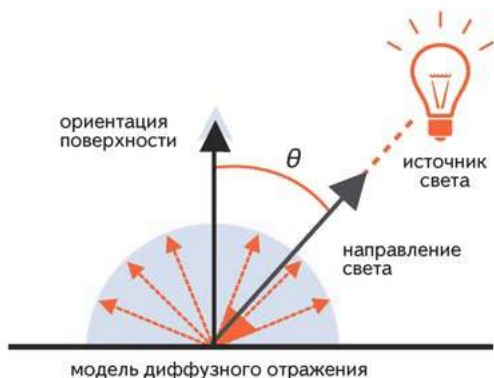


Рис. 8.6. Отраженный свет распространяется в направлении от поверхности к источнику света

Ориентация поверхности: что такое нормаль?

Ориентация поверхности определяется направлением перпендикуляра к ней, которое называется **нормалью**, или **вектором нормали**. Это направление представляется тремя числами, задающими вектор, направленный из точки в начале координат $(0, 0, 0)$ в точку (n_x, n_y, n_z) . Например, направление нормали $(1, 0, 0)$ совпадает с положительным направлением оси X, а направление нормали $(0, 0, 1)$ – с положительным направлением оси Z. Говоря о поверхностях и их нормалях, важно помнить два их свойства.

Поверхность имеет две нормали

Поскольку любая поверхность имеет две стороны, лицевую и изнаночную, каждая из сторон имеет свою нормаль; то есть, любая поверхность имеет две нормали. Например, лицевая сторона поверхности, перпендикулярной оси Z, имеет нормаль, направленную в сторону положительных значений Z, а изнаночная сторона – в сторону отрицательных, как показано на рис. 8.7. Эти нормали определяются как $(0, 0, 1)$ и $(0, 0, -1)$, соответственно.

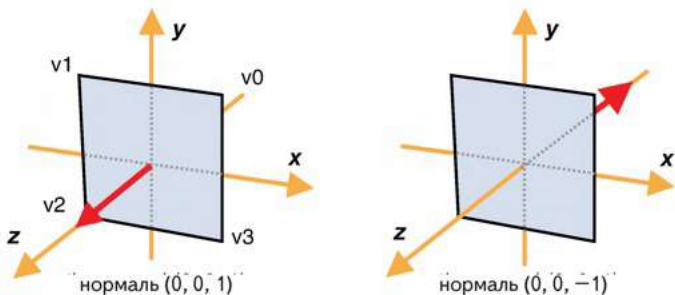


Рис. 8.7. Нормали

В трехмерной графике эти две стороны отличаются порядком, в каком указываются вершины при рисовании поверхности. Когда рисование поверхности осуществляется в порядке следования вершин⁷ v_0, v_1, v_2 и v_3 , лицевой стороной считается та, где вершины располагаются по часовой стрелке, если смотреть со стороны поверхности в направлении нормали (то же самое «правило правой руки», которое используется для определения положительного направления поворота, о котором рассказывалось в главе 3, «Рисование и преобразование треугольников»). Таким образом, на рис. 8.7 лицевая поверхность имеет нормаль $(0, 0, -1)$, она изображена справа.

Поверхности с одинаковой ориентацией имеют одну и ту же нормаль

Так как нормаль представляет только направление, поверхности с одинаковой ориентацией имеют одни и те же нормали, независимо от своих местоположений.

Если имеется несколько поверхностей с одинаковой ориентацией но в разных местоположениях, все они будут иметь одну и ту же нормаль. Например, поверхность, перпендикулярная к оси Z и с центром в точке с координатами $(10, 98, 9)$, будет иметь те же нормали $(0, 0, 1)$ и $(0, 0, -1)$. Эти же нормали будет иметь и поверхность с центром в точке начала координат (см. рис. 8.8).

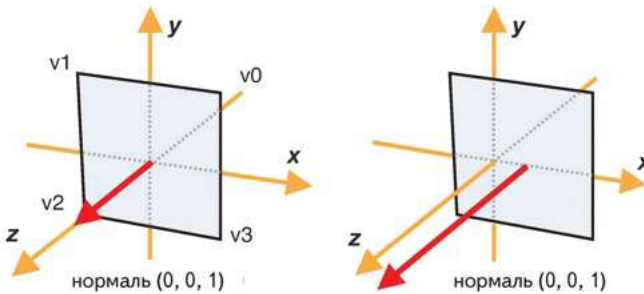


Рис. 8.8. Поверхности с одинаковой ориентацией имеют одни и те же нормали, независимо от своих местоположений

На рис. 8.9 слева показаны нормали, используемые в примерах программ в этом разделе. Они отмечены такими подписями, как « $n(0, 1, 0)$ ».

Следующей задачей после определения нормалей к поверхности является передача этих данных в шейдер. В предыдущей главе мы передавали информацию о цвете поверхности вместе с вершинами. Информацию о нормалях можно аналогичным способом: в буферном объекте вместе с вершинами. В этом разделе, как показано на рис. 8.9 (справа), информация о нормалях будет указываться для каждой вершины, а в этом случае каждой вершине соответствует три нормали, как и три цвета.⁸

⁷ В действительности данная поверхность состоит из двух треугольников: треугольника, который рисуется в порядке следования вершин v_0, v_1 и v_2 , и треугольника, который рисуется в порядке следования вершин v_0, v_2 и v_3 .

⁸ Куб или кубоид – это простой, но специфический объект, в каждой вершине которого сходятся три грани. Соответственно каждая вершина имеет три нормали. С другой стороны, гладкие объекты, такие как персонажи компьютерных игр, имеют по одной нормали на вершину.

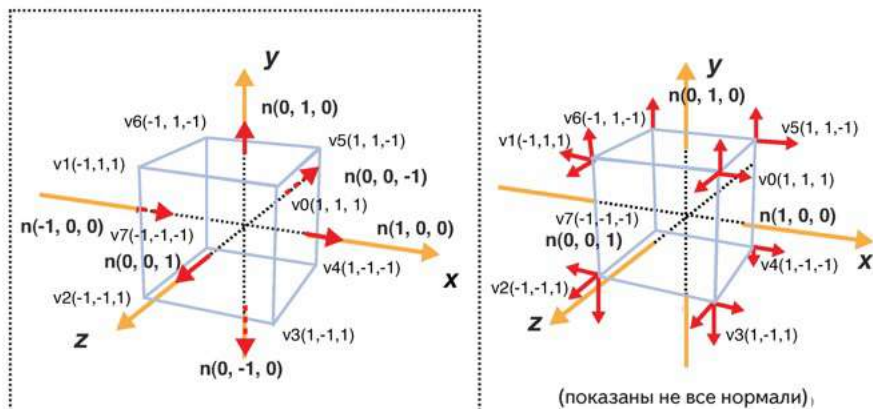
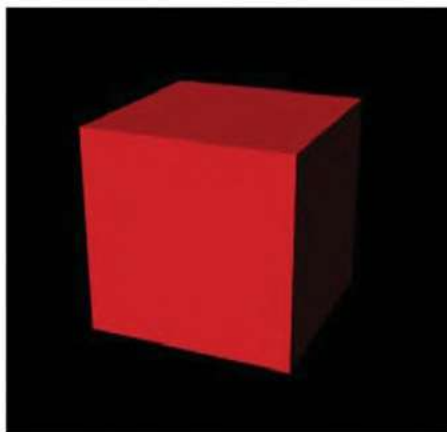


Рис. 8.9. Нормали к граням куба

Теперь перейдем к созданию программы `LightedCube`, отображающей красный куб, освещенный белым направленным светом. Результат работы программы показан на рис. 8.10.

Рис. 8.10. `LightedCube`

Пример программы (`LightedCube.js`)

Исходный код программы приводится в листинге 8.1. В ее основе лежит программа `ColoredCube` из предыдущей главы, поэтому структура этой программы в значительной мере совпадает с программой `ColoredCube`.

Как можно видеть из листинга 8.1, вершинный шейдер претерпел значительные изменения: в него добавилась реализация формулы 8.4. Кроме того, в функцию `initVertexBuffers()` добавлено определение информации о нормалях (строка 89), которая передается через переменную `a_Normal`. Фрагментный шейдер остался прежним, как в программе `ColoredCube`. Часть его оставлена в листинге,

чтобы вы могли убедиться, что он не выполняет никакой дополнительной обработки фрагментов.

Листинг 8.1. LightedCube.js

```

1 // LightedCube.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'attribute vec4 a_Color;\n' +
6   'attribute vec4 a_Normal;\n' +           // Нормаль
7   'uniform mat4 u_MvpMatrix;\n' +
8   'uniform vec3 u_LightColor;\n' +       // Цвет света
9   'uniform vec3 u_LightDirection;\n' +   // мировые координаты, нормализованные
10  'varying vec4 v_Color;\n' +
11  'void main() {\n' +
12  '  gl_Position = u_MvpMatrix * a_Position ;\n' +
13  '  // Нормализовать длину вектора нормали
14  '  vec3 normal = normalize(vec3(a_Normal));\n' +
15  '  // Скалярное произведение направления света на ориентацию поверхности
16  '  float nDotL = max(dot(u_LightDirection, normal), 0.0);\n' +
17  '  // Вычислить цвет в модели диффузного отражения
18  '  vec3 diffuse = u_LightColor * vec3(a_Color) * nDotL;\n' +
19  '  v_Color = vec4(diffuse, a_Color.a);\n' +
20  '}\n';
21
22 // Фрагментный шейдер
23 ...
24 'void main() {\n' +
25 '  gl_FragColor = v_Color;\n' +
26 '}\n';
27
28 function main() {
29   ...
30   // Определить координаты вершин, цвета и нормали
31   var n = initVertexBuffers(gl);
32   ...
33   var u_MvpMatrix = gl.getUniformLocation(gl.program, 'u_MvpMatrix');
34   var u_LightColor = gl.getUniformLocation(gl.program, 'u_LightColor');
35   var u_LightDirection = gl.getUniformLocation(gl.program, 'u_LightDirection');
36   ...
37   // Указать цвет света (белый)
38   gl.uniform3f(u_LightColor, 1.0, 1.0, 1.0);
39   // Определить направление света (в мировых координатах)
40   var lightDirection = new Vector3([0.5, 3.0, 4.0]);
41   lightDirection.normalize(); // Нормализовать
42   gl.uniform3fv(u_LightDirection, lightDirection.elements);
43
44   // Вычислить матрицу модели вида проекции
45   var mvpMatrix = new Matrix4(); // Матрица модели вида проекции
46   mvpMatrix.setPerspective(30, canvas.width/canvas.height, 1, 100);
47   mvpMatrix.lookAt(3, 3, 7, 0, 0, 0, 0, 1, 0);
48   // Передать матрицу в переменную u_MvpMatrix
49   gl.uniformMatrix4fv(u_MvpMatrix, false, mvpMatrix.elements);
50   ...

```



```

86  gl.drawElements(gl.TRIANGLES, n, gl.UNSIGNED_BYTE, 0); // Нарисовать куб
87  }
88
89  function initVertexBuffers(gl) {
    ...
98    var vertices = new Float32Array([ // Вершины
99      1.0, 1.0, 1.0, -1.0, 1.0, 1.0, -1.0, -1.0, 1.0, 1.0, -1.0, 1.0,
100     1.0, 1.0, 1.0, 1.0, -1.0, 1.0, 1.0, -1.0, -1.0, 1.0, 1.0, -1.0,
    ...
104     1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, 1.0, -1.0, 1.0, 1.0, -1.0
105   ]);
    ...
117
118   var normals = new Float32Array([ // Нормали
119     0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0,
120     1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0,
    ...
124     0.0, 0.0, -1.0, 0.0, 0.0, -1.0, 0.0, 0.0, -1.0, 0.0, 0.0, -1.0
125   ]);
    ...
140   if(!initArrayBuffer(gl, 'a_Normal', normals, 3, gl.FLOAT)) return -1;
    ...
154   return indices.length;
155 }

```

В качестве напоминания ниже еще раз приводится формула 8.4, реализованная в вершинном шейдере:

$$\begin{aligned}
 &\langle \text{цвет поверхности в модели диффузного отражения} \rangle = \\
 &\quad \langle \text{цвет света} \rangle \times \langle \text{собственный цвет поверхности} \rangle \times \\
 &\quad \langle \text{направление света} \rangle \cdot \langle \text{ориентация поверхности} \rangle
 \end{aligned}$$

Как видите, для вычислений по этой формуле необходимо иметь все четыре ее члена: (1) цвет испускаемого источником света, (2) собственный цвет поверхности, (3) направление на источник света и (4) ориентация поверхности. Кроме того, векторы, представляющие *направление света* и *ориентацию поверхности* должны быть нормализованы (иметь длину, равную 1.0).

Обработка в вершинном шейдере

Из четырех членов в формуле 8.4 собственный цвет поверхности передается в переменной `a_Color` (определяется в строке 5), как показано в следующем фрагменте, а ориентация поверхности передается в переменной `a_Normal` (определяется в строке 6). Цвет света передается в переменной `u_LightColor` (определяется в строке 8) и направление на источник света – в переменной `u_LightDirection` (определяется в строке 9). Обратите внимание, что только значение `u_LightDirection` передается в мировых координатах⁹ и нормализуется в

⁹ В этой книге все световые эффекты, связанные с затенением, вычисляются в системе мировых координат (см. приложение G, «Мировая и локальная системы координат»), потому что с ней проще работать и она выглядит более естественной для определения направления на источник света. Те же самые вычисления можно было произвести в системе видимых координат, но они выглядели бы более сложными.

программном коде JavaScript для простоты обработки. Это позволяет избежать необходимости нормализации в каждом вызове вершинного шейдера:

```

4  'attribute vec4 a_Position;\n' +
5  'attribute vec4 a_Color;\n' +           //          <- (2) цвет поверхности
6  'attribute vec4 a_Normal;\n' +         // Нормаль   <- (4) ориентация
7  'uniform mat4 u_MvpMatrix;\n' +
8  'uniform vec3 u_LightColor;\n' +       // Цвет света          <- (1)
9  'uniform vec3 u_LightDirection;\n' +   // мировые координаты <- (3)
10 'varying vec4 v_Color;\n' +
11 'void main() {\n' +
12 '    gl_Position = u_MvpMatrix * a_Position ;\n' +
13 '    // Нормализовать длину вектора нормали
14 '    vec3 normal = normalize(vec3(a_Normal));\n' +
15 '    // Скалярное произведение направления света на ориентацию поверхности
16 '    float nDotL = max(dot(u_LightDirection, normal), 0.0);\n' +
17 '    // Вычислить цвет в модели диффузного отражения
18 '    vec3 diffuse = u_LightColor * vec3(a_Color) * nDotL;\n' +
19 '    v_Color = vec4(diffuse, a_Color.a);\n' +
20 '}\n';

```

После получения всей необходимой информации можно приступить к вычислениям. Сначала вершинный шейдер нормализует вектор (строка 14). Технически, так как в этом примере нормали уже имеют длину 1.0, эта операция не является необходимой. Но это частный случай, поэтому мы решили выполнить нормализацию:

```

14 '    vec3 normal = normalize(vec3(a_Normal));\n' +

```

Хотя переменная `a_Normal` имеет тип `vec4`, нормали, как представляющие направления, определяются только компонентами `x`, `y` и `z`. Поэтому сначала приходится извлекать компоненты `.xyz` и затем нормализовать их. Если переменную для хранения нормали объявить с типом `vec3`, операция извлечения станет ненужной. Однако в данной программе она объявлена с типом `vec4`, потому что когда мы будем усовершенствовать код в следующем примере, нам потребуется именно переменная типа `vec4`. Но, подробнее об этом мы поговорим при обсуждении следующей программы. Как можно заметить, в языке GLSL ES имеется встроенная функция `normalize()`, реализующая нормализацию своего аргумента. Нормализованная нормаль сохраняется в переменной `normal` для последующего использования.

Затем нам нужно вычислить скалярное произведение *<направление света>* · *<ориентация поверхности>*. Направление на источник света хранится в переменной `u_LightDirection`. Так как это направление уже нормализовано, мы можем использовать его непосредственно, без промежуточной обработки. Ориентация поверхности хранится в переменной `normal` и уже была нормализована в строке 14. Скалярное произведение «·» можно вычислить с помощью функции `dot()`, которая также является встроенной функцией языка GLSL ES и возвращает скалярное произведение двух векторов, передаваемых ей в виде аргументов. То есть,

вызов `dot(u_LightDirection, normal)` вычислит $\langle \text{направление света} \rangle \cdot \langle \text{ориентация поверхности} \rangle$. Это вычисление производится в строке 16.

```
16 ' float nDotL = max(dot(u_LightDirection, normal), 0.0);\n' +
```

Если результат скалярного произведения является положительным числом, он присваивается переменной `nDotL`. Если отрицательным – переменной присваивается число 0.0. Здесь используется функция `max()`, еще одна встроенная функция GLSL ES, возвращающая наибольшее значение из двух аргументов.

Отрицательный результат скалярного произведения означает, что угол θ в выражении $\cos \theta$ имеет размер больше 90 градусов. А так как θ – это угол между направлением на источник света и ориентацией поверхности, его величина больше 90 градусов говорит о том, что свет падает на поверхность с изнаночной стороны (см. рис. 8.11), то есть свет не падает на лицевую сторону поверхности, соответственно переменной `nDotL` присваивается 0.0.

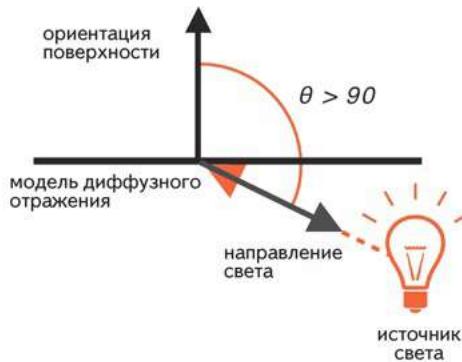


Рис. 8.11. Нормаль и свет в случае, когда угол θ больше 90 градусов

Теперь, когда подготовка закончена, по формуле 8.4 вычисляется требуемое значение цвета (строка 18). Значение переменной `a_Color`, которая имеет тип `vec4` и хранит цвет в формате RGBA, приводится к типу `vec3` (извлекаются компоненты `.rgb`), потому что значение прозрачности (альфа-канала) не участвует в вычислении цвета.

В действительности прозрачность оказывает существенное влияние на цвет поверхности. Однако, из-за того, что определение света, проникающего сквозь полупрозрачный объект, связано с достаточно сложными вычислениями, мы решили игнорировать значение прозрачности в этой программе:

```
18 ' vec3 diffuse = u_LightColor * vec3(a_Color) * nDotL;\n' +
```

По окончании вычислений, результат из переменной `diffuse` переписывается `varying`-переменную `v_Color` в строке 19. Так как `v_Color` имеет тип `vec4`, значение `diffuse` приводится к типу `vec4` со значением 1.0 в последнем компоненте:

```
19 ' v_Color = vec4(diffuse, a_Color.a);\n' +
```


Результат операций, описанных выше, – значение цвета, вычисленное с учетом нормали для вершины, – передается во фрагментный шейдер, где присваивается переменной `gl_FragColor`. В данном случае по условиям задачи куб освещается направленным светом, поэтому все вершины одной и той же поверхности имеют одинаковый цвет, то есть каждая грань имеет равномерную окраску, без теней и переходов.

На этом код вершинного шейдера и заканчивается. Далее мы посмотрим, как программа на JavaScript передает данные в вершинный шейдер, необходимые для вычислений по формуле 8.4.

Обработка в программе на JavaScript

Цвет света (`u_LightColor`) и направление на источник (`u_LightDirection`) передаются в вершинный шейдер из программного кода на JavaScript. Так как свет имеет белый цвет (1.0, 1.0, 1.0), мы просто записываем его в переменную `u_LightColor` с помощью метода `gl.uniform3f()`:

```
69 // Указать цвет света (белый)
70 gl.uniform3f(u_LightColor, 1.0, 1.0, 1.0);
```

Следующий шаг – определение направления на источник света, которое перед передачей необходимо еще и нормализовать, как уже отмечалось выше. Нормализовать его можно с помощью метода `normalize()` объекта `Vector3`, реализованного в библиотеке `cuon-matrix.js`. Этот метод достаточно прост в обращении: мы создаем объект `Vector3`, передав конструктору вектор, подлежащий нормализации (строка 72), и затем вызываем метод `normalize()` этого объекта. Не забывайте, что синтаксис JavaScript несколько отличается от синтаксиса GLSL ES:

```
71 // Определить направление света (в мировых координатах)
72 var lightDirection = new Vector3([0.5, 3.0, 4.0]);
73 lightDirection.normalize(); // Нормализовать
74 gl.uniform3fv(u_LightDirection, lightDirection.elements);
```

Результат записывается в свойство `elements` объекта типа `Float32Array` и затем присваивается переменной `u_LightDirection` вызовом `gl.uniform3fv()` (строка 74).

Наконец, в функции `initVertexBuffers()` (строка 89) выполняется запись информация о нормали. Фактическая информация о нормалях для каждой вершины хранится в массиве `normals`, который определяется в строке 118, вместе с информацией о цвете, как в программе `ColoredCube.js`. В строке 140 эта информация записывается в переменную `a_Normal` для передачи в вершинный шейдер:

```
140 if(!initArrayBuffer(gl,'a_Normal', normals, 3, gl.FLOAT)) return -1;
```

Функция `initArrayBuffer()`, которая также использовалась в программе `ColoredCube`, переписывает массив, переданный ей в третьем аргументе (`normals`), в переменную-атрибут, переданную во втором аргументе (`a_Normal`).

Добавление затенения, обусловленного фоновым освещением

Несмотря на то, что на этой стадии мы успешно справились с реализацией освещения сцены, куб, изображенный на рис. 8.9, все же немного отличается от того, что мы привыкли видеть в реальном мире. В частности, поверхность, находящаяся со стороны, противоположной источнику света, выглядит почти черной и плохо различима. Эту проблему проще заметить, если добавить анимационный эффект. Давайте напишем программу `LightedCube_animation` (см. рис. 8.12) чтобы увидеть проблему более отчетливо.

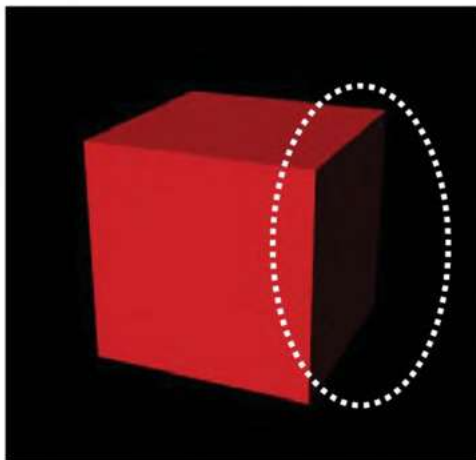


Рис. 8.12. Результат работы программы `LightedCube_animation`

Согласно формуле 8.4 сцена освещается правильно, но наш жизненный опыт говорит, что что-то в ней не так. Для нас несколько необычен такой резкий контраст – в реальном мире даже задние поверхности, обращенные от источника света, освещаются отраженным от других объектов светом. Фоновое освещение, упоминавшееся в предыдущем разделе, – это не прямое освещение и его можно использовать для придания сцене большей реалистичности. Давайте добавим это освещение в нашу сцену и посмотрим, действительно ли она станет выглядеть более реалистично. Так как в модели фонового отражения полагается, что свет падает на объект со всех направлений с одинаковой интенсивностью, цвет поверхности в отраженном свете определяется только цветом света и собственным цветом поверхности. Формула вычисления цвета для этого случая была ранее представлена как формула 8.2 и еще раз приводится ниже:

$$\begin{aligned} <\text{цвет поверхности в модели фонового отражения}> = \\ &<\text{цвет света}> \times <\text{собственный цвет поверхности}> \end{aligned}$$

Добавим в программу `LightedCube` определение цвета, обусловленного фоновым освещением, как описывает эта формула. Чтобы получить окончательный цвет, необходимо реализовать формулу 8.3:

$\langle \text{цвет поверхности при диффузном и фоновом отражении} \rangle =$
 $\langle \text{цвет в модели диффузного отражения} \rangle + \langle \text{цвет в модели фонового отражения} \rangle$

Фоновое освещение – слабое, потому что является результатом отражения света другими объектами, такими как стены. Например, если фоновый свет имеет цвет (0.2, 0.2, 0.2) а сама поверхность красная, или (1.0, 0.0, 0.0), тогда, согласно формуле 8.2, в модели фонового отражения поверхность получит цвет (0.2, 0.0, 0.0). Например, представьте белый кубик в комнате с синими стенами, то есть, поверхность кубика имеет собственный цвет (1.0, 1.0, 1.0), а фоновый свет имеет цвет (0.0, 0.0, 0.2) – он будет выглядеть слегка голубоватым (0.0, 0.0, 0.2).

Итак, реализуем эффект отражения фонового света в программе `LightedCube_ambient`, которая выведет куб, как показано на рис. 8.13. Как видите, поверхность, которая прежде не освещалась непосредственно, теперь стала немного ярче, благодаря чему изображение стало выглядеть более реалистично.

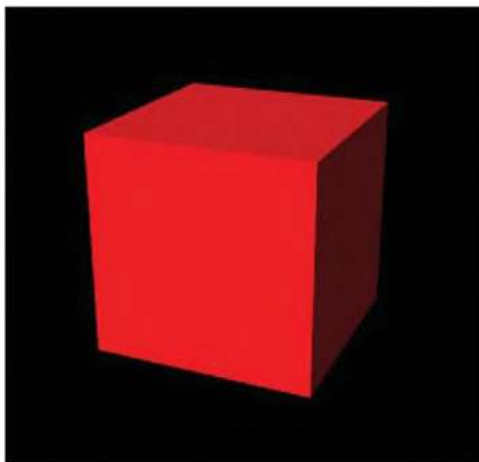


Рис. 8.13. `LightedCube_ambient`

Пример программы (`LightedCube_ambient.js`)

Исходный код программы приводится в листинге 8.2. Так как данная программа практически полностью повторяет программу `LightedCube`, в листинге показаны только изменившиеся фрагменты.

Листинг 8.2. `LightedCube_ambient.js`

```
1 // LightedCube_ambient.js
2 // Вершинный шейдер
3 ...
4 'uniform vec3 u_LightColor;\n' + // Цвет света
5 'uniform vec3 u_LightDirection;\n' + // мировые координаты, нормализованные
10 'uniform vec3 u_AmbientLight;\n' + // Цвет фонового света
11 'varying vec4 v_Color;\n' +
12 'void main() {\n' +
```

```

16     ...
17     float nDotL = max(dot(lightDirection, normal), 0.0);\n' +
18     // Вычислить цвет в модели диффузного отражения
19     ' vec3 diffuse = u_LightColor * a_Color.rgb * nDotL;\n' +
20     // Вычислить цвет в модели фонового отражения
21     ' vec3 ambient = u_AmbientLight * a_Color.rgb;\n' +
22     // Сложить цвет в модели диффузного и фонового отражения
23     ' v_Color = vec4(diffuse + ambient, a_Color.a);\n' +
24     ')\n';
25     ...
36 function main() {
26     ...
64     // Получить ссылки на uniform-переменные и так далее
27     ...
68     var u_AmbientLight = gl.getUniformLocation(gl.program, 'u_AmbientLight');
28     ...
80     // Указать цвет фонового света
81     gl.uniform3f(u_AmbientLight, 0.2, 0.2, 0.2);
29     ...
95 }

```

В вершинный шейдер добавлена переменная `u_AmbientLight` (строка 10) для передачи цвета фонового освещения. После вычислений по формуле 8.2 с использованием этого цвета и собственного цвета поверхности (`a_Color`), результат сохраняется в переменной `ambient` (строка 21). Теперь, когда определены значения обеих переменных, `diffuse` и `ambient`, в строке 23 вычисляется окончательный цвет поверхности по формуле 8.3. Результат передается в переменную `v_Color`, точно так же, как в приложении `LightedCube`, и поверхность окрашивается в этот цвет.

Как видите, эта программа просто добавляет цвет фонового освещения в строке 23, благодаря чему куб выглядит более освещенным. Эта программа реализует эффект освещения фоновым светом, падающим на объект со всех сторон.

В примерах, приводившихся до сих пор, демонстрировалась реализация освещения статических объектов. Однако программы трехмерной графики часто реализуют различные перемещения объектов в пределах сцены или изменение местоположения точки наблюдения, поэтому вы должны уметь обрабатывать подобные преобразования. Как вы уже знаете из главы 4, «Дополнительные преобразования и простая анимация», объект может перемещаться, масштабироваться и вращаться. Эти преобразования могут изменять направления нормалей и требовать пересчета освещенности при каждом изменении сцены. Давайте посмотрим, как реализовать это.

Освещенность перемещаемого и вращаемого объекта

Программа `LightedTranslatedRotatedCube` реализует поворот куба на 90 градусов по часовой стрелке вокруг оси Z и смещение на 0.9 единицы вдоль оси Y,

при этом предполагается, что куб освещается направленным светом. Подобно программе `LightedCube_ambient`, описанной в предыдущем разделе, данная программа так же учитывает освещенность от направленного источника света и освещенность фоновым светом. Результат показан на рис. 8.14.

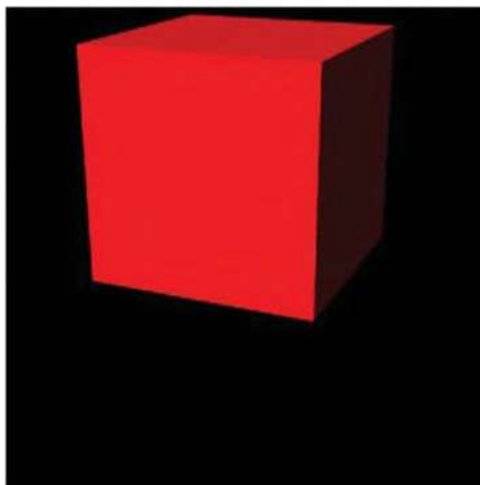


Рис. 8.14. `LightedTranslatedRotatedCube`

В предыдущем разделе мы видели, что направление нормали может изменяться при преобразованиях координат. На рис. 8.15 представлены некоторые примеры таких преобразований. На крайнем левом изображении на рис. 8.15 показано, как выглядит куб, который рисуется данной программой, если смотреть на него вдоль оси Z, в сторону отрицательных значений. Здесь показана только нормаль $(1, 0, 0)$, направленная в сторону положительных значений по оси X. Давайте выполним некоторые преобразования с этой фигурой (следующие три изображения на рис. 8.15).

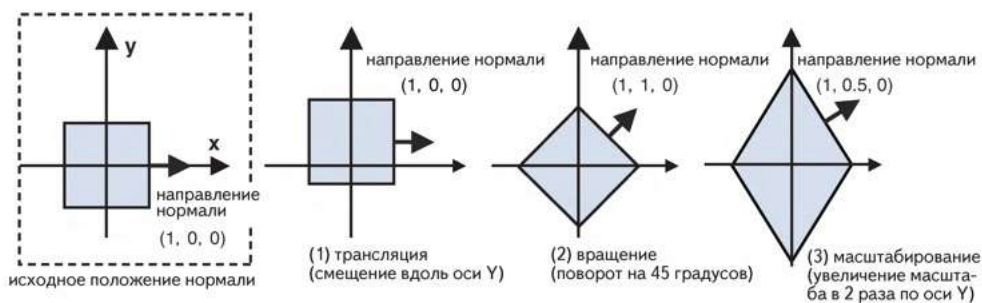


Рис. 8.15. Изменение направления нормали при различных преобразованиях координат

После изучения рис. 8.15 можно сделать следующие выводы:

- трансляция (перемещение) **не изменяет** направление нормали, поскольку при перемещении ориентация объекта не изменяется;
- вращение **изменяет** направление нормали, поскольку при повороте изменяется ориентация объекта;
- масштабирование оказывает более сложное влияние на направление нормали. Как показано на рисунке, объект сначала был повернут, а затем подвергся масштабированию с двукратным увеличением размеров по оси Y . В данном случае направление нормали **изменилось**, потому что изменилась ориентация поверхности. С другой стороны, если объект подвергнется масштабированию по всем осям с одинаковыми коэффициентами, направление нормали **не изменится**. Наконец, **даже если** подвергнется масштабированию с разными коэффициентами, направление нормали **может не измениться**. Например, если объект на крайнем левом изображении (исходное положение) увеличить в 2 раза только по оси Y , направление нормали не изменится.

Совершенно понятно, что вычисление направления нормали при разных трансформациях может оказаться довольно сложной задачей, особенно когда имеет место масштабирование. Однако, здесь нам на помощь может прийти математический аппарат.

Волшебство матриц: транспонированная обратная матрица

Как уже говорилось в главе 4, матрица, выполняющая преобразование координат объекта, называется матрицей модели. Направление нормали можно вычислить путем умножения нормали на **транспонированную обратную матрицу модели**. Под транспонированной обратной матрицей подразумевается матрица, являющаяся результатом транспонирования обратной матрицы.

Обратной для матрицы M является такая матрица R , для которой произведения $R \cdot M$ и $M \cdot R$ дают единичную матрицу. Под термином **транспонирование** подразумевается операция, меняющая строки и столбцы матрицы местами. Подробнее о транспонировании рассказывается в приложении Е, «Транспонированная обратная матрица». Для нас на данный момент достаточно ограничиться следующим правилом:

Примечание. Направление нормали можно вычислить, как произведение нормали на транспонированную обратную матрицу модели.

Транспонированная обратная матрица вычисляется следующим образом:

- выполняется обращение оригинальной матрицы;
- выполняется транспонирование полученной обратной матрицы;

Эти операции легко можно выполнить с помощью удобных методов объекта `Matrix4` (см. табл. 8.1).

Таблица 8.1. Методы объекта `Matrix4` для обращения и транспонирования матриц

Метод	Описание
<code>Matrix4.setInverseOf(m)</code>	Вычисляет обратную матрицу для матрицы <code>m</code> и сохраняет результат в объекте <code>Matrix4</code> . Параметр <code>m</code> также является объектом <code>Matrix4</code> .
<code>Matrix4.transpose()</code>	Транспонирует матрицу, хранящуюся в объекте <code>Matrix4</code> , и сохраняет результат в том же объекте <code>Matrix4</code> .

Если предположить, что матрица модели хранится в объекте `Matrix4` с именем `modelMatrix`, следующий фрагмент кода получает транспонированную обратную ей матрицу. Результат сохраняется в переменной `normalMatrix`, потому что выполняются преобразования координат нормали:

```
Matrix4 normalMatrix = new Matrix4();
// Вычислить матрицу модели
...
// Вычислить матрицу преобразований нормали в соответствии с матрицей модели
normalMatrix.setInverseOf(modelMatrix);
normalMatrix.transpose();
```

А теперь исследуем программу `LightedTranslatedRotatedCube.js`, реализующую освещение направленным светом куба, повернутого на 90 градусов по часовой стрелке относительно оси `Z` и смещенного на 0.9 единицы вдоль оси `Y`. Мы будем использовать куб из программы `LightedCube_ambient`.

Пример программы (*LightedTranslatedRotatedCube.js*)

Исходный код программы приводится в листинге 8.3. Отличие от программы `LightedCube_ambient` заключается в добавлении `u_NormalMatrix` (строка 8) для передачи в вершинный шейдер матрицы преобразований координат нормали. Преобразование выполняется в строке 16, с использованием матрицы `u_NormalMatrix`, вычисленной в JavaScript.

Листинг 8.3. `LightedTranslatedRotatedCube.js`

```
1 // LightedTranslatedRotatedCube.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4 ...
5
6 'attribute vec4 a_Normal;\n' +
7 'uniform mat4 u_MvpMatrix;\n' +
8 'uniform mat4 u_NormalMatrix;\n'+ // Матрица преобразования нормали
9 'uniform vec3 u_LightColor;\n' + // Цвет света
10 'uniform vec3 u_LightDirection;\n' + // Мировые координаты, нормализованные
11 'uniform vec3 u_AmbientLight;\n' + // Цвет фонового света
12 'varying vec4 v_Color;\n' +
```

```

13  'void main() {\n' +
14  '   gl_Position = u_MvpMatrix * a_Position;\n' +
15  '   // Вычислить нормаль и нормализовать ее
16  '   vec3 normal = normalize(vec3(u_NormalMatrix * a_Normal));\n' +
17  '   // Скалярное произведение направления света и нормали
18  '   float nDotL = max(dot(u_LightDirection, normal), 0.0);\n' +
19  '   // Вычислить цвет в модели диффузного отражения
20  '   vec3 diffuse = u_LightColor * a_Color.rgb * nDotL;\n' +
21  '   // Вычислить цвет в модели фонового отражения
22  '   vec3 ambient = u_AmbientLight * a_Color.rgb;\n' +
23  '   // Сложить цвет в модели диффузного и фонового отражения
24  '   v_Color = vec4(diffuse + ambient, a_Color.a);\n' +
25  ' }\n';
...
37 function main() {
...
65  // Получить ссылки на uniform-переменные и так далее
66  var u_MvpMatrix = gl.getUniformLocation(gl.program, 'u_MvpMatrix');
67  var u_NormalMatrix = gl.getUniformLocation(gl.program, 'u_NormalMatrix');
...
85  var modelMatrix = new Matrix4(); // Матрица модели
86  var mvpMatrix = new Matrix4(); // Матрица модели вида проекции
87  var normalMatrix = new Matrix4(); // Матрица преобразования нормали
88
89  // Вычислить матрицу модели
90  modelMatrix.setTranslate(0, 1, 0); // Перемещение по оси Y
91  modelMatrix.rotate(90, 0, 0, 1); // Поворот относительно оси Z
92  // Вычислить матрицу модели вида проекции
93  mvpMatrix.setPerspective(30, canvas.width/canvas.height, 1, 100);
94  mvpMatrix.lookAt(-7, 2.5, 6, 0, 0, 0, 0, 0, 1, 0);
95  mvpMatrix.multiply(modelMatrix);
96  // Передать матрицу в переменную u_MvpMatrix
97  gl.uniformMatrix4fv(u_MvpMatrix, false, mvpMatrix.elements);
98
99  // Вычислить матрицу преобразования нормали на основе матрицы модели
100 normalMatrix.setInverseOf(modelMatrix);
101 normalMatrix.transpose();
102 // Передать матрицу преобразования нормали в u_NormalMatrix
103 gl.uniformMatrix4fv(u_NormalMatrix, false, normalMatrix.elements);
...
110 }

```

Вершинный шейдер в этой программе действует почти так же, как в `LightedCube_ambient`. Отличия заключаются в строке, реализующей правило, представленное выше, согласно которому `a_Normal` нужно умножить на транспонированную обратную матрицу модели:

```

15  // Вычислить нормаль и нормализовать ее
16  '   vec3 normal = normalize(vec3(u_NormalMatrix * a_Normal));\n' +

```

Так как переменная `a_Normal` имеет тип `vec4`, ее можно умножить на матрицу `u_NormalMatrix` типа `mat4`. Однако нам нужно только компоненты `x`, `y` и `z` результата, поэтому он преобразуется в значение типа `vec3` вызовом `vec3()`. С этой

целью можно было бы использовать также `.xyz`, как это уже делалось прежде, и использовать форму записи `(u_NormalMatrix * a_Normal).xyz`. Однако для большей простоты мы использовали `vec3()`. Теперь, когда вы представляете, как шейдер вычисляет направление нормали после поворота и перемещения объекта, перейдем к программному коду на JavaScript. Особое внимание здесь следует обратить на вычисление матрицы, которая будет передана в вершинный шейдер через `u_NormalMatrix`.

Переменная `u_NormalMatrix` – это транспонированная обратная матрица модели, поэтому сначала вычисляется матрица модели (строки 90 и 91). Так как эта программа вращает объект относительно оси Z и транслирует (перемещает) его вдоль оси Y, мы можем использовать методы `setTranslate()` и `rotate()` объекта `Matrix4`, как описывалось в главе 4. В строках 100 и 101 выполняется обращение и транспонирование матрицы. В строке 103 полученный результат передается в вершинный шейдер через переменную `u_NormalMatrix`, так же как `mvpMatrix` в строке 97. Второй аргумент метода `gl.uniformMatrix4fv()` определяет необходимость транспонирования матрицы (см. главу 3):

```
99 // Вычислить матрицу преобразования нормали на основе матрицы модели
100 normalMatrix.setInverseOf(modelMatrix);
101 normalMatrix.transpose();
102 // Передать матрицу преобразования нормали в u_NormalMatrix
103 gl.uniformMatrix4fv(u_NormalMatrix, false, normalMatrix.elements);
```

После запуска, программа нарисует куб, как показано на рис. 8.14. Как видите затенение в этой программе выполнено так же, как в программе `LightedCube_ambient`, при этом куб смещен вдоль оси Y. Это объясняется тем, что (1) трансляция (перемещение) не изменяет направления нормалей, (2) точно так же направление нормалей не изменяет и поворот на 90 градусов, потому что поворот на этот угол просто меняет поверхности куба местами, (3) направление света от направленного источника не изменяется, независимо от местоположения объекта, и (4) в модели диффузного отражения отраженный свет распространяется во всех направлениях с одинаковой интенсивностью.

Теперь вы должны хорошо понимать основы реализации освещенности и затенения в трехмерной графике. Но не будем останавливаться на достигнутом и перейдем к знакомству с еще одним типом источников света: точечными источниками.

Освещение точечным источником света

В противоположность направленному освещению, освещенность от точечного источника света выглядит по-разному в разных точках трехмерной сцены (см. рис. 8.16). То есть, при вычислении цвета необходимо учитывать направление на источник света из каждой позиции на поверхности, куда падает свет.



Рис. 8.16. Направление на точечный источник света различается в разных точках

В предыдущих примерах мы вычисляли цвет для каждой вершины, передавая направление нормали и направление на источник света. Тот же подход мы будем использовать и здесь, но, так как направление на источник света теперь не является величиной постоянной, нам потребуется для каждой вершины передавать также местоположение источника света и затем вычислять направление на него.

Ниже представлен пример программы `PointLightedCube`, отображающей красный куб, освещаемый белым светом из точечного источника. Здесь мы снова реализовали модели диффузного и фонового отражения. Результат работы программы показан на рис. 8.17, которая фактически является версией программы `LightedCube_ambient` из предыдущего раздела, только на этот раз в ней реализовано освещение куба точечным источником света.

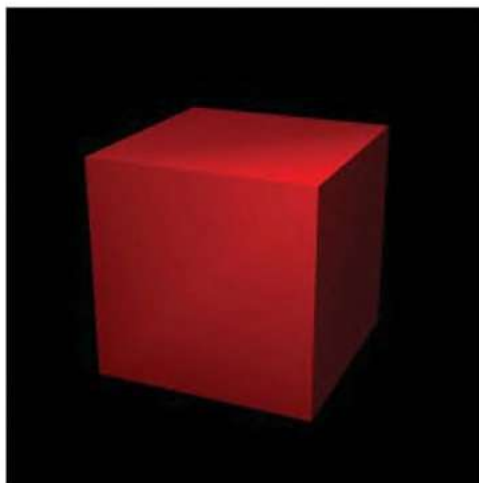


Рис. 8.17. `PointLightedCube`

Пример программы (*`PointLightedCube.js`*)

В листинге 8.4 представлен исходный код программы, в которой, в сравнении с `LightedCube_ambient`, изменилась только реализация вершинного шейдера. Мы

добавили переменную `u_ModelMatrix`, для передачи матрицы модели, и переменную `u_LightPosition`, представляющую местоположение источника света. Обратите внимание: так как в этой программе используется точечный источник света, вместо направления освещения мы будем использовать местоположение источника света. Кроме того, чтобы сделать эффект более заметным, мы увеличили размеры куба.

Листинг 8.4. PointLightedCube.js

```

1 // PointLightedCube.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4 'attribute vec4 a_Position;\n' +
5 ...
6
7 'uniform mat4 u_ModelMatrix;\n' + // Матрица модели
8 'uniform mat4 u_NormalMatrix;\n' + // Матрица преобразования нормали
9 'uniform vec3 u_LightColor;\n' + // Цвет света
10 'uniform vec3 u_LightPosition;\n' + // Позиция источника света
11 // (в мировых координатах)
12 'uniform vec3 u_AmbientLight;\n' + // Цвет фонового света
13 'varying vec4 v_Color;\n' +
14 'void main() {\n' +
15 '    gl_Position = u_MvpMatrix * a_Position;\n' +
16 '    // Нормализовать длину вектора нормали
17 '    vec3 normal = normalize(vec3(u_NormalMatrix * a_Normal));\n' +
18 '    // Найти мировые координаты вершины
19 '    vec4 vertexPosition = u_ModelMatrix * a_Position;\n' +
20 '    // Найти направление на источник света и нормализовать его
21 '    vec3 lightDirection = normalize(u_LightPosition - vec3(vertexPosition));\n' +
22 '    // Скалярное произведение направления света и нормали
23 '    float nDotL = max(dot(lightDirection, normal), 0.0);\n' +
24 '    // Вычислить цвет в модели диффузного отражения
25 '    vec3 diffuse = u_LightColor * a_Color.rgb * nDotL;\n' +
26 '    // Вычислить цвет в модели фоновое отражения
27 '    vec3 ambient = u_AmbientLight * a_Color.rgb;\n' +
28 '    // Сложить цвет в модели диффузного и фоновое отражения
29 '    v_Color = vec4(diffuse + ambient, a_Color.a);\n' +
30 '}\n';
31
32 ...
33
34 function main() {
35     ...
36     // Получить ссылки на uniform-переменные и так далее
37     var u_ModelMatrix = gl.getUniformLocation(gl.program, 'u_ModelMatrix');
38     ...
39     var u_LightColor = gl.getUniformLocation(gl.program, 'u_LightColor');
40     var u_LightPosition = gl.getUniformLocation(gl.program, 'u_LightPosition');
41     ...
42     // Указать цвет света (белый)
43     gl.uniform3f(u_LightColor, 1.0, 1.0, 1.0);
44     // Указать позицию источника света (в мировых координатах)
45     gl.uniform3f(u_LightPosition, 0.0, 3.0, 4.0);
46     ...

```

```

89  var modelMatrix = new Matrix4(); // Матрица модели
90  var mvpMatrix = new Matrix4();   // Матрица модели вида проекции
91  var normalMatrix = new Matrix4(); // Матрица преобразования нормали
92
93  // Вычислить матрицу модели
94  modelMatrix.setRotate(90, 0, 1, 0); // Поворот относительно оси Y
95  // Передать матрицу модели в переменную u_ModelMatrix
96  gl.uniformMatrix4fv(u_ModelMatrix, false, modelMatrix.elements);
...

```

Основными отличиями данного вершинного шейдера являются строки с 19 по 21. В строке 19 координаты вершины преобразуются в мировые координаты, чтобы затем вычислить направление на источник света. Так как точечный источник испускает свет во всех направлениях, направлением на него в вершине является результат вычитания местоположения вершины из местоположения источника света. А поскольку местоположение источника передается в переменной `u_Light-Position` (объявляется в строке 11) в мировых координатах, чтобы определить направление, координаты вершины также требуется преобразовать в мировые. Далее, в строке 21, вычисляется направление на источник света. Обратите внимание, что направление нормализуется вызовом `normalize()`, чтобы привести длину вектора к 1.0. Используя направление на источник света (`lightDirection`), шейдер вычисляет скалярное произведение (строка 23), чтобы получить цвет поверхности для текущей вершины.

Если запустить эту программу, мы получим довольно реалистичное изображение, как показано на рис. 8.17. Но, несмотря на неплохую реалистичность, при ближайшем рассмотрении на изображении можно заметить некоторые недостатки: линия тени на поверхностях куба выглядит недостаточно естественно (см. рис. 8.18). Эта неестественность более заметна при вращении куба, которое мы реализовали в программе `PointLightedCube_animation`.

Этот недостаток обусловлен выполнением интерполяции, обсуждавшейся в главе 5, «Цвет и текстура». Как вы наверняка помните, система WebGL интерполирует значение цвета для точек, между вершинами, исходя из цвета, назначенного для вершин. Однако, из-за того, что во всех точках поверхности направление на источник света разное, для более естественного затенения цвет нужно вычислять для каждой точки, а не только для вершин. Эта проблема проявляется еще сильнее с использованием сфер, как показано на рис. 8.19.

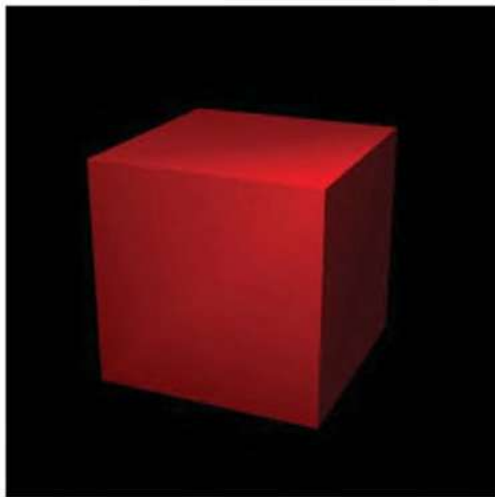


Рис. 8.18. Неестественность при вычислении освещенности для каждой вершины

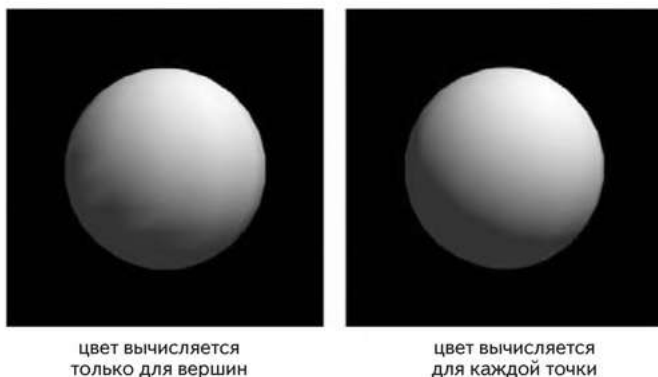


Рис. 8.19. Сферы, освещаемые точечным источником света

Как видите, граница между освещенной и неосвещенной частями на изображении слева выглядит неестественной. Если вам трудно заметить описываемый эффект на рисунке в книге, воспользуйтесь нашими программами `PointLightedSphere` (создает изображение слева) и `PointLightedSphere_perFragment` (создает изображение справа). В следующем разделе мы расскажем, как устранить данную проблему.

Более реалистичное затенение: вычисление цвета для каждого фрагмента

На первый взгляд кажется, что вычисление цвета для каждой точки на поверхности куба, куда падает свет, — практически нереализуемая задача. Однако, по сути это означает вычисление цвета для фрагментов и включение в работу фрагментного шейдера.

Эта программа называется `PointLightedCube_perFragment`, и результат ее работы показан на рис. 8.20.

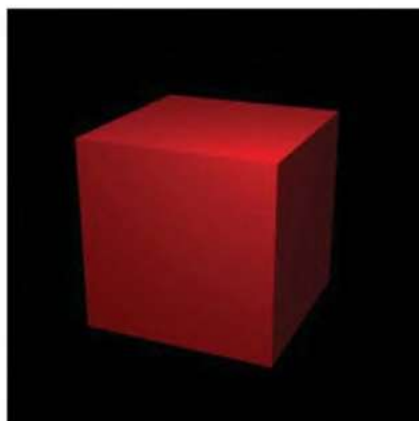


Рис. 8.20. `PointLightedCube_perFragment`

Пример программы (PointLightedCube_perFragment.js)

В листинге 8.5 приводится пример программы, основанной на PointLightedCube.js. В ней изменился только код шейдеров и, как нетрудно заметить, объем вычислений в вершинном шейдере уменьшился, а во фрагментном – увеличился.

Листинг 8.5. PointLightedCube_perFragment.js

```

1 // PointLightedCube_perFragment.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   ...
6   'uniform mat4 u_ModelMatrix;\n' + // Матрица модели
7   'uniform mat4 u_NormalMatrix;\n' + // Матрица преобразования нормали
8   'varying vec4 v_Color;\n' +
9   'varying vec3 v_Normal;\n' +
10  'varying vec3 v_Position;\n' +
11  'void main() {\n' +
12    '  gl_Position = u_MvpMatrix * a_Position;\n' +
13    '  // Найти мировые координаты вершины
14    '  v_Position = vec3(u_ModelMatrix * a_Position);\n' +
15    '  v_Normal = normalize(vec3(u_NormalMatrix * a_Normal));\n' +
16    '  v_Color = a_Color;\n' +
17  '}\n';
18
19 // Фрагментный шейдер
20 var FSHADER_SOURCE =
21   ...
22   'uniform vec3 u_LightColor;\n' + // Цвет света
23   'uniform vec3 u_LightPosition;\n' + // Позиция источника света
24   'uniform vec3 u_AmbientLight;\n' + // Цвет фонового света
25   'varying vec3 v_Normal;\n' +
26   'varying vec3 v_Position;\n' +
27   'varying vec4 v_Color;\n' +
28   'void main() {\n' +
29     // Нормализовать нормаль, которая интерполируется и не равна 1.0 (длина)
30     '  vec3 normal = normalize(v_Normal);\n' +
31     // Вычислить направление на источник света и нормализовать
32     '  vec3 lightDirection = normalize(u_LightPosition - v_Position);\n' +
33     // Скалярное произведение направления света и нормали
34     '  float nDotL = max(dot(lightDirection, normal), 0.0);\n' +
35     // Вычислить окончательный цвет с применением моделей
36     // диффузного и фонового отражения
37     '  vec3 diffuse = u_LightColor * v_Color.rgb * nDotL;\n' +
38     '  vec3 ambient = u_AmbientLight * v_Color.rgb;\n' +
39     '  gl_FragColor = vec4(diffuse + ambient, v_Color.a);\n' +
40   '}\n';

```

Чтобы вычислить цвет для каждого фрагмента, необходимо (1) знать местоположение фрагмента в мировых координатах и (2) направление нормали в позиции

фрагмента. Для получения этих значений можно использовать процедуру интерполяции (глава 5), вычисляя исходные значения для вершин в вершинном шейдере и передавая их через *varying*-переменные во фрагментный шейдер.

Эти вычисления выполняются в строках 16 и 17. В строке 16 определяются мировые координаты вершины, путем умножения вектора координат вершины на матрицу модели. После сохранения координат вершины в *varying*-переменной *v_Position*, ее значение будет интерполироваться для точек между вершинами и передаваться в соответствующей переменной (*v_Position*) во фрагментный шейдер. Расчеты нормали в строке 17 преследуют ту же цель.¹⁰ После сохранения результата в переменной *v_Normal*, она так же будет интерполироваться и передаваться через переменную *v_Normal* во фрагментный шейдер.

Во фрагментном шейдере выполняются те же операции, что прежде выполнялись в вершинном шейдере, в программе *PointLightedCube.js*. Во-первых, в строке 34, выполняется нормализация интерполированной нормали, переданной из вершинного шейдера, так как ее длина может оказаться не равной 1.0 из-за интерполяции. Далее, в строке 36, вычисляется и нормализуется направление на источник света. На основании этих результатов, в строке 38, вычисляется скалярное произведение направления света и нормали. Значения цвета в моделях диффузного и фонового отражения вычисляется в строках 40 и 41, и в строке 42 они складываются, чтобы получить окончательный цвет фрагмента, который тут же присваивается переменной *gl_FragColor*.

Если имеется более одного источника света, после вычисления цвета в моделях диффузного и фонового отражения для каждого источника, окончательный цвет фрагмента получается путем сложения цветов. Иными словами, вычисления по формуле 8.3 могут выполняться сколько угодно раз, в зависимости от числа источников цвета.

В заключение

В этой главе мы исследовали реализацию освещения трехмерной сцены, разные типы освещения и познакомились с двумя моделями отражения света. Затем, используя эти знания, мы попробовали реализовать освещение трехмерных объектов разными источниками света и исследовали некоторые приемы затенения, повышающие реалистичность изображений. Как вы могли заметить, владение искусством моделирования освещения позволяет добавить значительную долю реализма в трехмерные сцены, которые иначе могут казаться плоскими и неинтересными.

¹⁰ В данном примере программы нормализация не требуется, потому что все нормали, что передаются в переменной *a_Normal*, уже нормализованы. Однако мы оставили операцию нормализации, чтобы сделать наш код более универсальным.



ГЛАВА 9.

Иерархические объекты

Эта глава завершает описание основных особенностей и приемов программирования с применением WebGL. К концу этой главы вы овладеете всеми основами WebGL и будете в состоянии самостоятельно создавать реалистичные и интерактивные трехмерные сцены. Основное внимание в этой главе уделяется иерархическим объектам, играющим важную роль, потому что позволяют перейти от простых объектов, таких как кубы и блоки, к более сложным конструкциям, которые можно использовать в роли персонажей игр, роботов и даже моделировать на их основе людей.

Ниже перечислены основные темы, обсуждаемые в этой главе:

- моделирование сложных конструкций, таких как рука-манипулятор, с использованием иерархических структур;
- рисование и управление иерархическими объектами, состоящими из нескольких простых объектов;
- объединение матрицы модели с матрицей вращения для имитации таких сочленений, как суставы;
- внутренняя реализация функции `initShaders()`, которую мы уже использовали, но пока не исследовали.

К концу этой главы вы получите все знания, необходимые для создания трехмерных сцен, наполненных простыми и сложными трехмерными объектами.

Рисование составных объектов и управление ими

До сих пор мы рассказывали, как перемещать и вращать единственный объект, такой как плоский треугольник или объемный куб. Но многие объекты в трехмерной компьютерной графике, такие как персонажи компьютерных игр, роботы и так далее, состоят из множества объектов (или сегментов). В качестве простого примера можно привести руку-манипулятор, изображенную на рис. 9.1. Нетрудно заметить, что она состоит из множества параллелепипедов. Программа, реализующая это изображение, называется `MultiJointModel`. Давайте сначала загрузим ее и поэкспериментируем, нажимая клавиши со стрелками, **x**, **z**, **c** и **v**, чтобы понять, что нам предстоит сконструировать в следующих разделах.



←→: arm1 rotation, ↑ ↓: joint1 rotation, xz: joint2(wrist) rotation, cv: finger rotation

Рис. 9.1. Рука-манипулятор состоит из множества объектов

Одной из ключевых проблем, связанных с рисованием составных объектов, является предотвращение конфликтов при перемещении сегментов. В этом разделе мы исследуем данную проблему и расскажем, как нарисовать руку-манипулятор, состоящую из множества объектов (сегментов) и как управлять ею. Для начала рассмотрим, как устроена рука человека, от плеча до кончиков пальцев, чтобы понять, как смоделировать руку-манипулятор. Рука состоит из нескольких сегментов, таких как плечо, предплечье, кисть и пальцы, сочлененных друг с другом посредством суставов, как показано на рис. 9.2 слева.

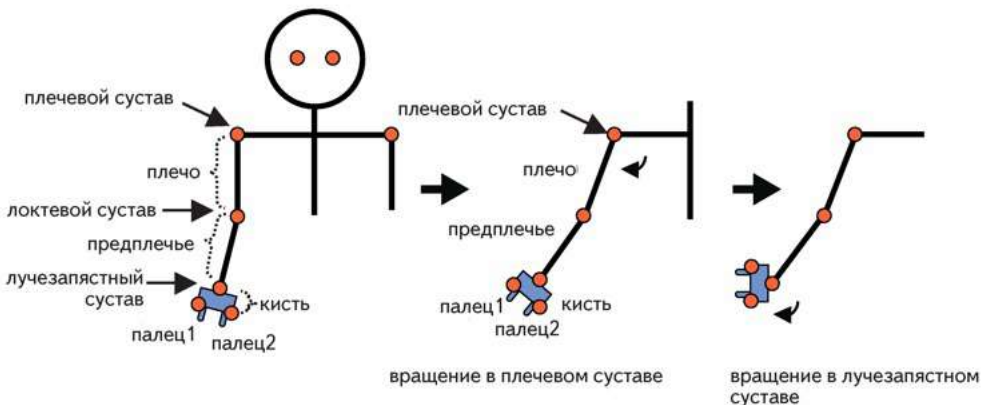


Рис. 9.2. Устройство руки и доступные движения

Каждый сегмент вращается в точке сочленения, как описывается ниже:

- при перемещении плеча вращение происходит в плечевом суставе, предплечье, кисть и пальцы двигаются вместе с плечом (как показано в середине на рис. 9.2 в середине);

- при перемещении предплечья вращение происходит в локтевом суставе, кисть и пальцы двигаются вместе с ним, но плечо остается на месте;
- при перемещении кисти вращение происходит в лучезапястном суставе, кисть и пальцы двигаются, а плечо с предплечьем – нет (как показано на рис. 9.2 справа);
- при перемещении пальцев плечо, предплечье и кисть остаются неподвижными.

Итак, когда двигается какой-то сегмент, сегменты, находящиеся ниже, перемещаются вместе с ним, а сегменты выше остаются неподвижными. Кроме того, все перемещения, включая вращения, фактически происходят за счет вращения относительно сустава.

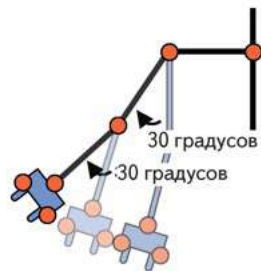
Иерархическая структура

Обычно рисование объектов с такими свойствами и управление ими осуществляется по отдельности (сегментами), в порядке иерархической структуры, от верхних к нижним, и к каждому сегменту применяется своя матрица модели (матрица вращения), вычисленная для каждого сочленения (сустава). Например, как показано на рис. 9.2, для всех суставов – плечевого, локтевого, лучезапястного и суставов пальцев – определяются отдельные матрицы вращения.

Важно отметить, что в отличие от конечностей человека или робота, сегменты в трехмерной графике физически не связаны между собой. Поэтому, если по невнимательности повернуть только плечо в плечевом суставе, нижние сегменты останутся на месте, из-за чего нарушится целостность восприятия. Поэтому, вращая плечо в плечевом суставе, необходимо явно переместить остальные сегменты руки. Для этого нужно повернуть локтевой и лучезапястный суставы на тот же угол, что и плечевой сустав.

Реализовать вращение сегмента так, чтобы оно распространялось на нижние сегменты, совсем несложно. Для этого достаточно использовать одну и ту же матрицу модели для вращения нижних сегментов. Например, при выполнении поворота в плечевом суставе на 30 градусов, одну и ту же матрицу модели можно использовать для поворота локтевого и лучезапястного суставов на те же 30 градусов (см. рис. 9.3). При таком подходе, изменяя угол поворота в плечевом суставе, нижние сегменты автоматически будут поворачиваться, следуя за движением плеча.

В более сложных случаях, когда после поворота в плечевом суставе на 30 градусов требуется выполнить поворот в локтевом суставе еще на 10 градусов, последнее вращение можно выполнить с применением матрицы модели, описывающей поворот на угол, больше угла поворота в плечевом суставе на 10 градусов. Вычислить эту матрицу можно, умножив матрицу модели поворота в плечевом суставе



вращение в плечевом суставе

Рис. 9.3. Нижние сегменты следуют за вращением верхнего сегмента

на матрицу вращения, описывающей поворот на 10 градусов. Результат можно в данном случае назвать «матрицей модели, описывающей поворот в локтевом суставе». Сегменты, расположенные ниже локтевого сустава, будут следовать за движением локтя, при рисовании с использованием этой «локтевой» матрицы модели.

При таком подходе верхние сегменты не будут затрагиваться вращением нижних сегментов. То есть, верхние сегменты не будут перемещаться, независимо от того, насколько большим будет поворот нижних сегментов.

Теперь, после знакомства с основными принципами перемещения объектов, состоящих из нескольких сегментов, перейдем к изучению примера программы.

Модель с единственным сочленением

Начнем с простой модели, включающей единственное сочленение. Напишем программу `JointModel`, рисующую руку-манипулятор, которая состоит из двух сегментов. Управление манипулятором осуществляется клавишами со стрелками. Скриншот и иерархическая структура показаны на рис. 9.4, слева и справа, соответственно. Рука-манипулятор состоит из сегмента1 и сегмента2, соединенных сочленением1. Рука на рис. 9.4 вытянута вверх и сегмент1 является верхней частью руки (плечом), а сегмент2 – нижней (предплечьем). Когда позднее мы добавим кисть, такая раскладка станет понятнее.

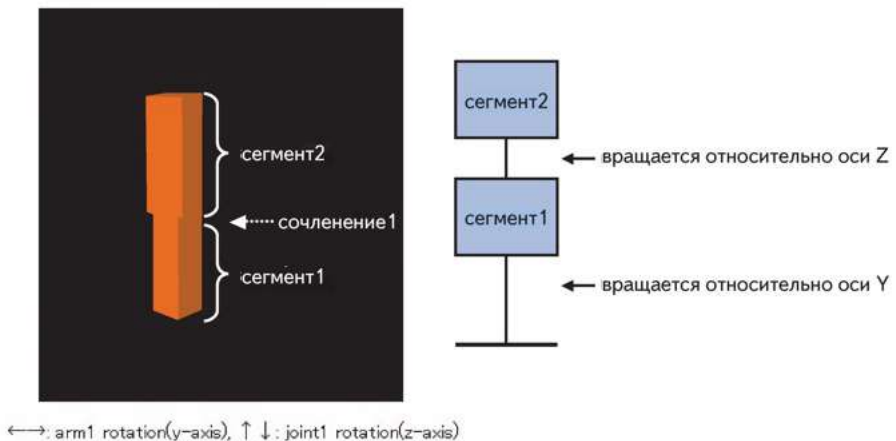
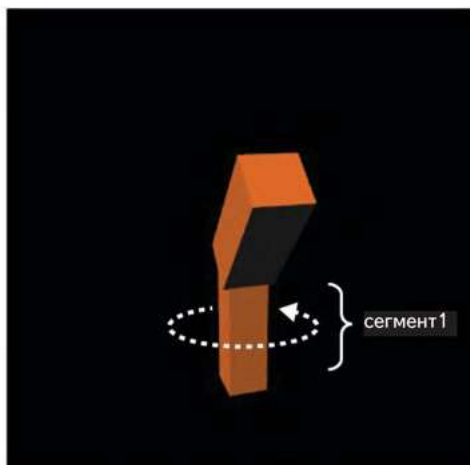


Рис. 9.4. `JointModel` и иерархическая структура, используемая в программе

Если запустить программу, можно увидеть, что сегмент1 вращается относительно оси Y клавишами со стрелками влево и вправо, а сочленение1 вращается относительно оси Z клавишами со стрелками вверх и вниз (рис. 9.5). При нажатии на клавишу со стрелкой вниз произойдет поворот в сочленении1 и сегмент2 наклонится вперед, как показано на рис. 9.5 слева. Если затем нажать клавишу со стрелкой вправо, сегмент1 повернется, как показано на рис. 9.5 справа.



←→: arm1 rotation(y-axis), ↑ ↓: joint1 rotation(z-axis)



←→: arm1 rotation(y-axis), ↑ ↓: joint1 rotation(z-axis)

Рис. 9.5. Результат нажатия клавиш со стрелками в программе JointModel

Как видите, перемещение сегмента2, вызванное вращением в сочленении1, не оказывает влияния на положение сегмента1. Напротив, перемещение сегмента1 вызывает также перемещение сегмента2.

Пример программы (JointModel.js)

В листинге 9.1 приводится исходный код программы JointModel.js. Код вершинного шейдера в действительности немного сложнее – ради экономии места мы убрали из листинга код, реализующий затенение. Однако, если вам интересно, как мы применили знания, полученные в предыдущих разделах этой главы, загляните в полный исходный код программы, которую можно загрузить с веб-сайта книги. В этой программе используется источник направленного света и несколько упрощена реализация модели диффузного отражения, что сделало сцену более объемной на вид. Однако мы не использовали никаких специальных вычислений для этой модели руки-манипулятора и весь код, необходимый для рисования и управления ею, написан на JavaScript.

Листинг 9.1. JointModel.js

```
1 // JointModel.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'attribute vec4 a_Normal;\n' +
6   'uniform mat4 u_MvpMatrix;\n' +
7   ...
8   'void main() {\n' +
9     '  gl_Position = u_MvpMatrix * a_Position;\n' +
10     '  // Расчет затенения, чтобы придать руке-манипулятору объемный вид
```

```

...
17  '}\n';
...
29 function main() {
    ...
46  // Определить координаты вершин.
47  var n = initVertexBuffers(gl);
    ...
57  // Получить ссылки на uniform-переменные
58  var u_MvpMatrix = gl.getUniformLocation(gl.program, 'u_MvpMatrix');
59  var u_NormalMatrix = gl.getUniformLocation(gl.program, 'u_NormalMatrix');
    ...
65  // Вычислить матрицу вида проекции
66  var viewProjMatrix = new Matrix4();
67  viewProjMatrix.setPerspective(50.0, canvas.width/canvas.height, 1.0, 100.0);
68  viewProjMatrix.lookAt(20.0, 10.0, 30.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
69
70  // Зарегистрировать обработчик событий нажатий на клавиши
71  document.onkeydown = function(ev){ keydown(ev, gl, n, viewProjMatrix,
                                     u_MvpMatrix, u_NormalMatrix); };
72  // Нарисовать руку-манипулятор
73  draw(gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix);
74 }
75
76 var ANGLE_STEP = 3.0;    // Шаг изменения угла поворота (в градусах)
77 var g_arm1Angle = 90.0;  // Угол поворота сегмента1 (в градусах)
78 var g_joint1Angle = 0.0; // Угол поворота сочленения1 (в градусах)
79
80 function keydown(ev, gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix) {
81     switch (ev.keyCode) {
82         case 38: // стрелка вверх -> увеличение угла поворота сочленения1 (ось Z)
83             if (g_joint1Angle < 135.0) g_joint1Angle += ANGLE_STEP;
84             break;
85         case 40: // стрелка вниз -> уменьшение угла поворота сочленения1 (ось Z)
86             if (g_joint1Angle > -135.0) g_joint1Angle -= ANGLE_STEP;
87             break;
88         ...
91         case 37: // стрелка влево -> уменьшение угла поворота сегмента1 (ось Y)
92             g_arm1Angle = (g_arm1Angle - ANGLE_STEP) % 360;
93             break;
94         default: return;
95     }
96     // Нарисовать руку-манипулятор
97     draw(gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix);
98 }
99
100 function initVertexBuffers(gl) {
101     // Координаты вершин
102     ...
148 }
    ...
174 // Матрица преобразования координат
175 var g_modelMatrix = new Matrix4(), g_mvpMatrix = new Matrix4();
176

```



```

177 function draw(gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix) {
    ...
181 // Сегмент1
182 var armLength = 10.0; // длина сегмента1
183 g_modelMatrix.setTranslate(0.0, -12.0, 0.0);
184 g_modelMatrix.rotate(g_armAngle, 0.0, 1.0, 0.0); // Поворот, ось Y
185 drawBox(gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix); // Рисовать
186
187 // Сегмент2
188 g_modelMatrix.translate(0.0, armLength, 0.0); // сместить в точку сочленения1
189 g_modelMatrix.rotate(g_jointAngle, 0.0, 0.0, 1.0); // Поворот, ось Z
190 g_modelMatrix.scale(1.3, 1.0, 1.3); // Сделать чуть тоньше
191 drawBox(gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix); // Рисовать
192 }
193
194 var g_normalMatrix = new Matrix4(); // Матрица преобразования нормали
195
196 // Нарисовать куб
197 function drawBox(gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix) {
198 // Вычислить матрицу модели вида проекции и передать в u_MvpMatrix
199 g_mvpMatrix.set(viewProjMatrix);
200 g_mvpMatrix.multiply(g_modelMatrix);
201 gl.uniformMatrix4fv(u_MvpMatrix, false, g_mvpMatrix.elements);
202 // Вычислить матрицу преобразования нормали и передать в u_NormalMatrix
203 g_normalMatrix.setInverseOf(g_modelMatrix);
204 g_normalMatrix.transpose();
205 gl.uniformMatrix4fv(u_NormalMatrix, false, g_normalMatrix.elements);
206 // Нарисовать
207 gl.drawElements(gl.TRIANGLES, n, gl.UNSIGNED_BYTE, 0);
208 }

```

Функция `main()`, определение которой начинается в строке 29, имеет ту же структуру, что и в примерах программ, демонстрировавшихся прежде. Первые существенные отличия начинаются в функции `initVertexBuffers()`, которая вызывается в строке 47. В функции `initVertexBuffers()` информация о вершинах сегмента1 и сегмента2 записываются в соответствующие буферные объекты. До сих пор мы рисовали кубы с длиной стороны 2.0 и центром в начале координат. Теперь, для большей реалистичности, каждый сегмент рисуется как кубоид (параллелепипед), изображенный на рис. 9.6 слева. Кубоид располагается так, что центр его основания совпадает с началом координат, и имеет размеры $3.0 \times 3.0 \times 10.0$. Благодаря установке центра основания кубоида в начало координат, его вращение относительно оси Z выполняется так же, как вращение сочленения1 (см. рис. 9.5), что помогло упростить программный код. Оба сегмента руки-манипулятора рисуются с использованием определения этого кубоида.

В строках с 66 по 68 вычисляется матрица вида проекции (`viewProjMatrix`) для указанного видимого объема, местоположения точки наблюдения и направления взгляда.

Управление перемещением руки-манипулятора в этой программе осуществляется с клавиатуры, в строке, поэтому в строке 71 регистрируется обработчик событий `keydown()`:

```

70 // Зарегистрировать обработчик событий нажатий на клавиши
71 document.onkeydown = function(ev) { keydown(ev, gl, n, viewProjMatrix,
                                     u_MvpMatrix, u_NormalMatrix); };
72 // Нарисовать руку-манипулятор
73 draw(gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix);

```

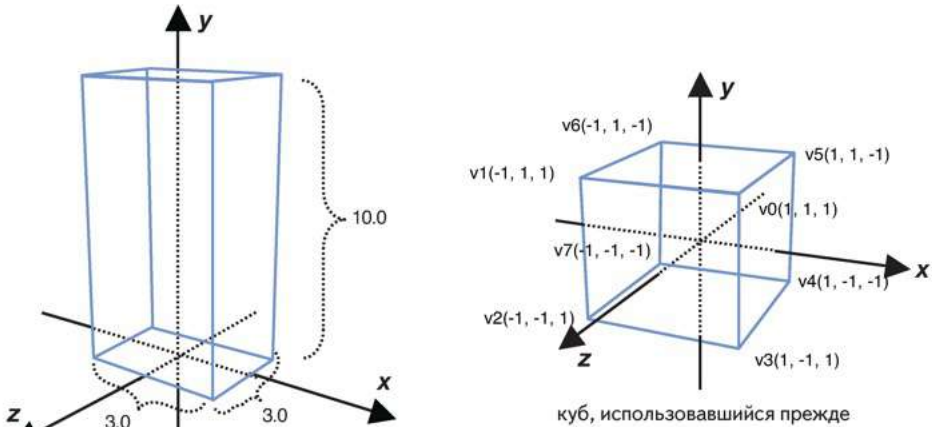


Рис. 9.6. Кубоид для рисования сегментов руки-манипулятора

Определение самой функции `keydown()` начинается в строке 80. Перед ним, в строках 76, 77 и 78, определяются глобальные переменные, используемые в `keydown()`:

```

76 var ANGLE_STEP = 3.0; // Шаг изменения угла поворота (в градусах)
77 var g_arm1Angle = 90.0; // Угол поворота сегмента1 (в градусах)
78 var g_joint1Angle = 0.0; // Угол поворота сочленения1 (в градусах)
79
80 function keydown(ev, gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix) {
81     switch (ev.keyCode) {
82         case 38: // стрелка вверх -> увеличение угла поворота сочленения1 (ось Z)
83             if (g_joint1Angle < 135.0) g_joint1Angle += ANGLE_STEP;
84             break;
85         ...
86         case 39: // стрелка вправо -> увеличение угла поворота сегмента1 (ось Y)
87             g_arm1Angle = (g_arm1Angle + ANGLE_STEP) % 360;
88             break;
89         ...
90     }
91 }
92 // Нарисовать руку-манипулятор
93 draw(gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix);
94 }

```

Переменная `ANGLE_STEP` (строка 76) определяет величину изменения угла поворота сегмента1 и сочленения1 при каждом нажатии на клавиши со стрелками, которая равна 3.0 градусам. Переменные `g_arm1Angle` (строка 77) и `g_joint1Angle` (строка 78) хранят текущие углы поворота сегмента1 и сочленения1, соответственно (см. рис. 9.7).

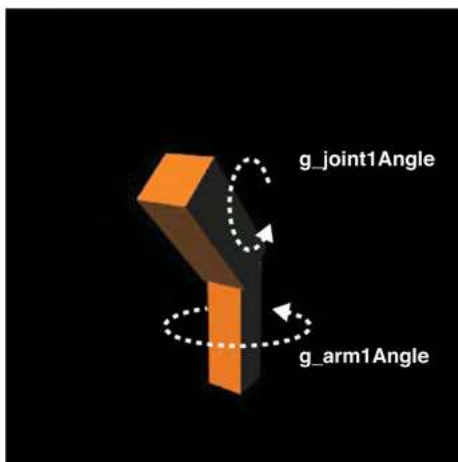


Рис. 9.7. Переменные `g_arm1Angle` и `g_joint1Angle` хранят текущие углы поворота

Функция `keydown()`, в зависимости от нажатой клавиши, увеличивает или уменьшает угол поворота сегмента1 (`g_arm1Angle`) или сочленения1 (`g_joint1Angle`) на значение `ANGLE_STEP`. Сочленение1 может вращаться только в пределах от -135 до 135 градусов, чтобы сегмент2 не сталкивался с сегментом1. Затем, в строке 97, производится рисование руки-манипулятора вызовом функции `draw()`.

Рисование иерархической структуры (`draw()`)

Функция `draw()`, определение которой начинается в строке 177, рисует руку-манипулятор в соответствии с заданной. В строке 175 определяются две глобальные переменные, `g_modelMatrix` и `g_mvpMatrix`, используемые обеими функциями, `draw()` и `drawBox()`:

```

174 // Матрица преобразования координат
175 var g_modelMatrix = new Matrix4(), g_mvpMatrix = new Matrix4();
176
177 function draw(gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix) {
178     ...
181     // Сегмент1
182     var armLength = 10.0; // длина сегмента1
183     g_modelMatrix.setTranslate(0.0, -12.0, 0.0);
184     g_modelMatrix.rotate(g_arm1Angle, 0.0, 1.0, 0.0); // Поворот, ось Y
185     drawBox(gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix); // Рисовать
186
187     // Сегмент2
188     g_modelMatrix.translate(0.0, armLength, 0.0); // сместить в точку сочленения1
189     g_modelMatrix.rotate(g_joint1Angle, 0.0, 0.0, 1.0); // Поворот, ось Z
190     g_modelMatrix.scale(1.3, 1.0, 1.3); // Сделать чуть тоньше
191     drawBox(gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix); // Рисовать
192 }

```


Как видите, функция `draw()` рисует сегменты с помощью `drawBox()`, начиная с верхнего сегмента (сегмент1) и заканчивая нижним (сегмент2).

При рисовании каждого сегмента повторяется одна и та же последовательность операций: (1) трансляция (`setTranslate()`, `translate()`), (2) вращение (`rotate()`) и (3) рисование сегмента (`drawBox()`).

При рисовании иерархических моделей, для которых выполняется вращение, обработка обычно выполняется сверху вниз, в порядке (1) трансляция, (2) вращение и (3) рисование.

Сегмент1 переносится в координаты (0.0, -12.0, 0.0) вызовом `setTranslate()` в строке 183, чтобы вся модель уместилась в видимом объеме. Так как этот сегмент вращается относительно оси Y, его матрица модели (`g_modelMatrix`) умножается в строке 184 на матрицу вращения, описывающую поворот относительно оси Y. Здесь используется переменная `g_arm1Angle`. По завершении преобразований координат сегмента1 вызывается функция `drawBox()`, чтобы нарисовать его.

Сегмент2 сочленен с сегментом1, как показано на рис. 9.7, поэтому он должен рисоваться, начиная с позиции, где заканчивается сегмент1. Добиться этого можно, сместив сегмент2 вдоль оси Y в направлении положительных значений на длину сегмента1 (`arm1Length`), применив матрицу преобразований, которая использовалась при рисовании сегмента1 (`g_modelMatrix`).

Эта операция выполняется в строке 188, где вторым аргументом функции `translate()` передается переменная `arm1Length`. Обратите также внимание, что вместо `setTranslate()` используется метод `translate()`, потому что сегмент2 рисуется с позиции, где заканчивается сегмент1:

```
187 // Сегмент2
188 g_modelMatrix.translate(0.0, arm1Length, 0.0); // сместить в точку сочленения1
189 g_modelMatrix.rotate(g_joint1Angle, 0.0, 0.0, 1.0); // Поворот, ось Z
190 g_modelMatrix.scale(1.3, 1.0, 1.3); // Сделать чуть тоньше
191 drawBox(gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix); // Рисовать
```

Строка 189 осуществляет поворот сегмента2, где, как можно видеть, используется переменная `g_joint1Angle`. В строке 190 размеры сегмента2 по осям X и Y немного уменьшаются с помощью операции масштабирования. Это помогает визуально различать два сегмента, но не оказывает никакого влияния на реализацию их перемещения.

Теперь, обновляя значения переменных `g_arm1Angle` и `g_joint1Angle` в функции `keydown()`, как описывалось в предыдущем разделе, и вызывая `draw()`, мы можем осуществлять вращение сегмента1 на угол `g_arm1Angle` и сегмента2 на угол `g_arm1Angle` и дополнительно на угол `g_joint1Angle`.

Функция `drawBox()` более чем проста. В строках 199 и 200 она вычисляет матрицу модели вида проекции и передает ее в переменную `u_MvpMatrix`. Затем, исходя из матрицы модели, она вычисляет матрицу преобразования координат нормали для затенения, сохраняет ее в переменной `u_NormalMatrix` (строки 203 и 204) и в строке 207 рисует кубоид, изображенный на рис. 9.6.

Этот базовый подход можно использовать для рисования сколь угодно сложных иерархических моделей, просто повторяя последовательность операций, описанных выше.

Очевидно, что наша рука-манипулятор, хотя и является моделью руки человека, больше похожа на изображение скелета руки, чем на настоящую руку. Для большей реалистичности ее можно было бы «обтянуть» кожей, однако эта тема далеко выходит за рамки данной книги, поэтому мы не будем обсуждать ее. Желающие могут обратиться за дополнительной информацией к книге «OpenGL ES 2.0 Programming Guide».

Модель со множеством сочленений

В этом разделе мы создадим программу `MultiJointModel`, расширенную версию `JointModel`, которая рисует руку-манипулятор с несколькими сочленениями – плечо, предплечье, кисть и два пальца – каждым из которых можно управлять с клавиатуры. Как показано на рис. 9.8, сегмент1 манипулятора закреплен на станине, сегмент2 соединен с сегментом1 с помощью сочленения1. Со свободным концом сегмента2, через сочленение2, соединена кисть. И со свободным концом кисти соединены два пальца – палец1 и палец2.

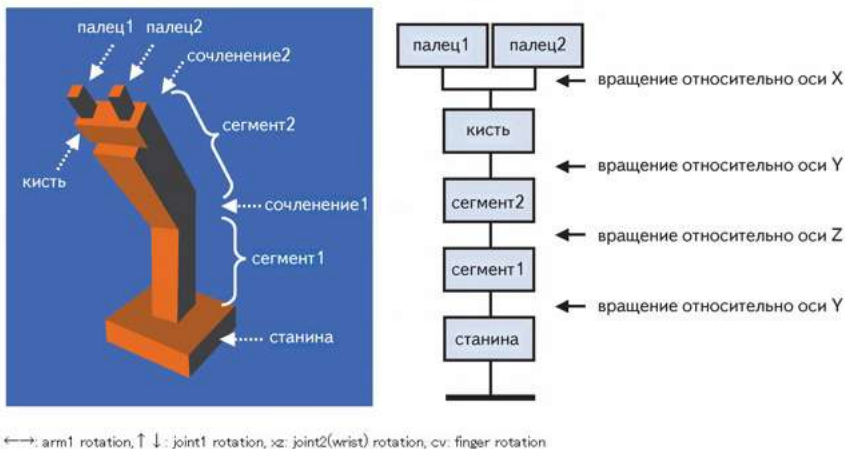


Рис. 9.8. Иерархическая структура модели в программе `MultiJointModel`

Управление сегментом1 и сочленением1 осуществляется клавишами со стрелками, как в программе `JointModel`. Кроме того, в данной программе есть возможность выполнять вращение в сочленении2 (лучезапястный сустав) клавишами **x** и **z**, и перемещать (вращать) два пальца клавишами **c** и **v**. Переменные, управляющие углом поворота каждой части руки-манипулятора, показаны на рис. 9.9.

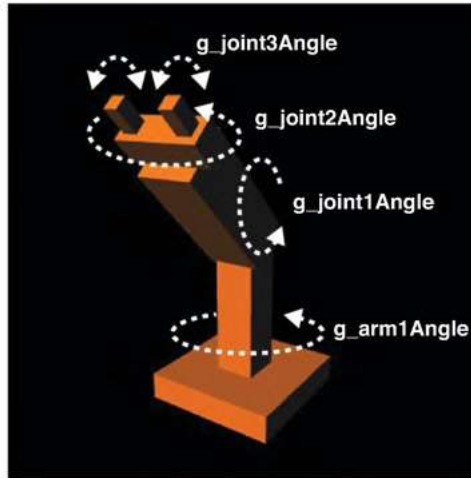


Рис. 9.9. Переменные, управляющие углом поворота каждой части руки-манипулятора

Пример программы (MultiJointModel.js)

Эта программа похожа на программу JointModel, за исключением расширенной функции `keydown()`, обрабатывающей дополнительные клавиши, и `draw()`, реализующей рисование расширенной иерархической структуры. Сначала рассмотрим функцию `keydown()`, представленную в листинге 9.2.

Листинг 9.2. MultiJointModel.js (код обработки нажатий на клавиши)

```
1 // MultiJointModel.js
...
76 var ANGLE_STEP = 3.0; // Шаг изменения угла поворота (в градусах)
77 var g_arm1Angle = 90.0; // Угол поворота сегмента1 (в градусах)
78 var g_joint1Angle = 45.0; // Угол поворота сочленения1 (в градусах)
79 var g_joint2Angle = 0.0; // Угол поворота сочленения2 (в градусах)
80 var g_joint3Angle = 0.0; // Угол поворота сочленения3 (в градусах)
81
82 function keydown(ev, gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix) {
83   switch (ev.keyCode) {
84     case 40: // стрелка вверх -> увеличение угла поворота сочленения1 (ось Z)
85       ...
86       break;
87     case 90: // клавиша Z -> увеличение угла поворота сочленения2
88       g_joint2Angle = (g_joint2Angle + ANGLE_STEP) % 360;
89       break;
90     case 88: // клавиша X -> уменьшение угла поворота сочленения2
91       g_joint2Angle = (g_joint2Angle - ANGLE_STEP) % 360;
92       break;
93     case 86: // клавиша V -> увеличение угла поворота сочленения3
94       if (g_joint3Angle < 60.0) g_joint3Angle = (g_joint3Angle +
```

```

104         break;
105     case 67: // клавиша C -> the уменьшение угла поворота сочленения3
106         if (g_joint3Angle > -60.0) g_joint3Angle = (g_joint3Angle -
107             ↗ANGLE_STEP) % 360;
108         break;
109     default: return;
110 }
111 // Нарисовать руку-манипулятор
112 draw(gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix);
113 }

```

Функция `keydown()` в основе своей осталась прежней, что и в программе `JointAngle`, но, кроме управления значениями переменных `g_arm1Angle` и `g_joint1Angle`, в нее была добавлена обработка клавиш **z**, **x**, **v** и **c**, (строки 96, 99, 102 и 105). Нажатие на эти клавиши изменяет переменные `g_joint2Angle` (хранит угол поворота сочленения2) и `g_joint3Angle` (хранит угол поворота сочленения3). После изменения переменных вызывается функция `draw()` (в строке 111), чтобы нарисовать иерархическую структуру. Рассмотрим теперь функцию `draw()`, представленную в листинге 9.3.

Несмотря на то, что для рисования всех элементов – станины, сегмента1, сегмента2, кисти, пальца1 и пальца2 – используется один и тот же кубоид, все они имеют разные размеры. Чтобы упростить рисование элементов, была расширена функция `drawBox()`, принимающая теперь на три аргумента больше:

```

function drawBox(gl, n, width, height, depth, viewProjMatrix, u_MvpMatrix,
    ↗u_NormalMatrix)

```

Передавая параметры `width`, `height` и `depth`, мы можем с помощью этой функции рисовать кубоиды указанных размеров с центром нижнего основания в начале координат.

Листинг 9.3. MultiJointModel.js (код рисования иерархической структуры)

```

188 // Матрица преобразования координат
189 var g_modelMatrix = new Matrix4(), g_mvpMatrix = new Matrix4();
190
191 function draw(gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix) {
192     // Очистить буферы цвета и глубины
193     gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
194
195     // Нарисовать станину
196     var baseHeight = 2.0;
197     g_modelMatrix.setTranslate(0.0, -12.0, 0.0);
198     drawBox(gl, n, 10.0, baseHeight, 10.0, viewProjMatrix, u_MvpMatrix,
199         ↗u_NormalMatrix);
200
201     // Сегмент1
202     var arm1Length = 10.0;
203     g_modelMatrix.translate(0.0, baseHeight, 0.0); // переместить на станину
204     g_modelMatrix.rotate(g_arm1Angle, 0.0, 1.0, 0.0); // Повернуть

```

```

204 drawBox(gl, n, 3.0, armLength, 3.0, viewProjMatrix, u_MvpMatrix,
    ↪u_NormalMatrix); // Нарисовать
205
206 // Сегмент2
207 ...
212 // Кисть
213 var palmLength = 2.0;
214 ...
218 // Переместить к свободному концу кисти
219 g_modelMatrix.translate(0.0, palmLength, 0.0);
220
221 // Нарисовать палец1
222 pushMatrix(g_modelMatrix);
223 g_modelMatrix.translate(0.0, 0.0, 2.0);
224 g_modelMatrix.rotate(g_joint3Angle, 1.0, 0.0, 0.0); // Повернуть
225 drawBox(gl, n, 1.0, 2.0, 1.0, viewProjMatrix, u_MvpMatrix, u_NormalMatrix);
226 g_modelMatrix = popMatrix();
227
228 // Нарисовать палец2
229 g_modelMatrix.translate(0.0, 0.0, -2.0);
230 g_modelMatrix.rotate(-g_joint3Angle, 1.0, 0.0, 0.0); // Повернуть
231 drawBox(gl, n, 1.0, 2.0, 1.0, viewProjMatrix, u_MvpMatrix, u_NormalMatrix);
232 }
233
234 var g_matrixStack = []; // Массив для хранения матриц
235 function pushMatrix(m) { // Сохраняет указанную матрицу в массив
236     var m2 = new Matrix4(m);
237     g_matrixStack.push(m2);
238 }
239
240 function popMatrix() { // Извлекает матрицу из массива
241     return g_matrixStack.pop();
242 }

```

Функция `draw()` действует точно так же, как в программе `JointModel`; то есть, она выполняет операции в следующем порядке: (1) трансляция, (2) поворот и (3) рисование (вызовом `drawBox()`). Прежде всего, поскольку станина не вращается, после перемещения ее в соответствующую позицию (строка 197), функция `draw()` вызывает `drawBox()`, чтобы нарисовать ее. В аргументах с третьего по пятый функции `drawBox()` передаются ширина (10), высота (2) и толщина (10), соответствующие плоской станине.

Сегмент1, сегмент2 и кисть рисуются с помощью той же последовательности операций: (1) трансляция, (2) поворот и (3) рисование, в порядке элементов от верхних элементов к нижним, как в программе `JointModel`.

Основное отличие этой программы заключается в рисовании пальцев, начиная со строки 222. Так как они не связаны отношением родитель-потомок, при работе с ними требуется проявить особую внимательность. В частности, необходимо уделить внимание содержимому матрицы модели. Сначала рассмотрим палец1, который перемещается в позицию 2.0 вдоль оси Z и поворачивается относительно оси X. Палец1 можно нарисовать с помощью такой последовательности операций:

(1) трансляция, (2) поворот и (3) рисование, как и прежде. В программе эта последовательность реализована так:

```
g_modelMatrix.translate(0.0, 0.0, 2.0);  
g_modelMatrix.rotate(g_joint3Angle, 1.0, 0.0, 0.0); // Поворот  
drawBox(gl, n, 1.0, 2.0, 1.0, u_MvpMatrix, u_NormalMatrix);
```

Теперь рассмотрим палец2. Если рисовать его с помощью той же последовательности операций, возникнет проблема. Палец2 нужно переместить в позицию -2.0 вдоль оси Z и повернуть относительно оси X. Однако, из-за того, что матрица модели изменилась, если просто нарисовать палец2, он окажется присоединен к концу пальца1.

Очевидно, что проблему можно решить, восстановив матрицу модели в состояние, предшествовавшее рисованию пальца1. Проще всего добиться желаемого – сохранить матрицу модели перед рисованием пальца1 и восстановить ее перед рисованием пальца2. Именно это и делают строки 222 и 226, где вызываются функции `pushMatrix()` и `popMatrix()`. В строке 222, вызовом `pushMatrix()`, выполняется сохранение указанной матрицы модели (`g_modelMatrix`). Затем, после рисования пальца1 в строках с 223 по 225, вызовом функции `popMatrix()` выполняется восстановление прежней матрицы модели в переменной `g_modelMatrix`. Теперь, благодаря восстановлению матрицы модели, можно нарисовать палец2, используя ту же последовательность операций, что и прежде.

Функции `pushMatrix()` и `popMatrix()` показаны ниже. Функция `pushMatrix()` сохраняет указанную матрицу в массиве с именем `g_matrixStack`. Функция `popMatrix()` извлекает матрицу из массива `g_matrixStack` и возвращает ее:

```
234 var g_matrixStack = []; // Массив для хранения матриц  
235 function pushMatrix(m) { // Сохраняет указанную матрицу в массив  
236     var m2 = new Matrix4(m);  
237     g_matrixStack.push(m2);  
238 }  
239  
240 function popMatrix() { // Извлекает матрицу из массива  
241     return g_matrixStack.pop();  
242 }
```

Этот подход можно использовать для рисования сколь угодно сложной руки-манипулятора. Он легко распространяется на любое число дополнительных элементов в иерархии. Вам нужно будет только вызывать `pushMatrix()` и `popMatrix()` в точках рисования «братских» элементов, не связанных отношением родитель-потомок.

Рисование сегментов (`drawBox()`)

А теперь перейдем к функции `drawBox()`, которая рисует сегменты руки-манипулятора, используя следующие аргументы:

```
247 function drawBox(gl, n, width, height, depth, viewMatrix, u_MvpMatrix,  
    u_NormalMatrix) {
```

Аргументы с третьего по пятый – `width`, `height` и `depth` – определяют ширину, высоту и толщину кубоида. Что касается остальных аргументов: `viewMatrix` – это матрица вида, а `u_MvpMatrix` и `u_NormalMatrix` – это ссылки на `uniform`-переменные внутри вершинного шейдера, посредством которых передаются матрицы преобразования координат, точно так же, как в программе `JointModel.js`. Матрица модели вида проекции передается в переменную `u_MvpMatrix`, а матрица преобразования координат нормали – в переменную `u_NormalMatrix`.

В отличие от `JointModel`, используемый здесь трехмерный объект – это куб со стороной 1.0. Центр нижней его грани находится в начале координат, благодаря чему вращение сегментов руки-манипулятора реализуется очень просто. Ниже показана функция `drawBox()`:

```

244 var g_normalMatrix = new Matrix4();// матрица преобразования нормали
245
246 // Рисует кубоид
247 function drawBox(gl, n, width, height, depth, viewProjMatrix,
                u_MvpMatrix, u_NormalMatrix) {
248     pushMatrix(g_modelMatrix); // Сохранить матрицу модели
249     // Масштабировать куб и нарисовать
250     g_modelMatrix.scale(width, height, depth);
251     // Вычислить матрицу модели вида проекции и передать в u_MvpMatrix
252     g_mvpMatrix.set(viewProjMatrix);
253     g_mvpMatrix.multiply(g_modelMatrix);
254     gl.uniformMatrix4fv(u_MvpMatrix, false, g_mvpMatrix.elements);
255     // Вычислить матрицу преобразования нормали и передать в u_NormalMatrix
256     ...
259     // Нарисовать
260     gl.drawElements(gl.TRIANGLES, n, gl.UNSIGNED_BYTE, 0);
261     g_modelMatrix = popMatrix(); // Восстановить матрицу модели
262 }
```

Как видите, матрица модели умножается на матрицу масштабирования (строка 250), поэтому куб будет нарисован в виде параллелепипеда с размерами, указанными в аргументах `width`, `height` и `depth`. Обратите внимание, как в строках 248 и 261 осуществляется сохранение и восстановление матрицы модели вызовом функций `pushMatrix()` и `popMatrix()`. Если этого не сделать, когда программа начнет рисование сегмента2 после сегмента1, масштаб, использованный при рисовании сегмента1, останется в матрице модели и окажет влияние на форму сегмента2. Сохраняя матрицу модели (строка 248) и восстанавливая ее (строка 261), мы возвращаемся к матрице модели, которая имела место перед масштабированием в строке 250.

Да, необходимость использования функций `pushMatrix()` и `popMatrix()` добавляет сложности в программу, но это позволяет использовать единственный комплект координат вершин и применять масштабирование для создания разных кубоидов. Как вариант, можно было бы также использовать множество объектов разных размеров. Давайте посмотрим, как это можно реализовать.

Рисование сегментов (drawSegment())

В этом разделе мы расскажем, как рисовать сегменты, переключаясь между буферными объектами, которые хранят координаты вершин всех сегментов. Обычно для каждого сегмента требуется определить координаты вершин, нормали и индексы. Однако в этом примере, так как все сегменты являются кубоидами, мы можем использовать единый для всех сегментов комплект нормалей и индексов, и определить для каждого из них только координаты вершин. Координаты вершин сегментов (станина, сегмент1, сегмент2, кисть и пальцы) хранятся в отдельных буферных объектах, а программа, представленная в листинге 9.4, просто переключается между ними.

Листинг 9.4. MultiJointModel_segment.js

```
1 // MultiJointModel_segment.js
...
29 function main() {
...
47   var n = initVertexBuffers(gl);
...
57   // Получить ссылки на переменные-атрибуты и uniform-переменные
58   var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
...
74   draw(gl, n, viewProjMatrix, a_Position, u_MvpMatrix, u_NormalMatrix);
75 }
...
115 var g_baseBuffer = null;    // Буферный объект для станины
116 var g_arm1Buffer = null;    // Буферный объект для сегмента1
117 var g_arm2Buffer = null;    // Буферный объект для сегмента2
118 var g_palmBuffer = null;    // Буферный объект для кисти
119 var g_fingerBuffer = null;  // Буферный объект для пальцев
120
121 function initVertexBuffers(gl){
122   // Координаты вершин (для всех сегментов)
123   var vertices_base = new Float32Array([ // Станина (10x2x10)
124     5.0, 2.0, 5.0, -5.0, 2.0, 5.0, -5.0, 0.0, 5.0, 5.0, 0.0, 5.0,
125     5.0, 2.0, 5.0, 5.0, 0.0, 5.0, 5.0, 0.0, -5.0, 5.0, 2.0, -5.0,
...
129     5.0, 0.0, -5.0, -5.0, 0.0, -5.0, -5.0, 2.0, -5.0, 5.0, 2.0, -5.0
130   ]);
131
132   var vertices_arm1 = new Float32Array([ // Сегмент1 (3x10x3)
133     1.5, 10.0, 1.5, -1.5, 10.0, 1.5, -1.5, 0.0, 1.5, 1.5, 0.0, 1.5,
134     1.5, 10.0, 1.5, 1.5, 0.0, 1.5, 1.5, 0.0, -1.5, 1.5, 10.0, -1.5,
...
138     1.5, 0.0, -1.5, -1.5, 0.0, -1.5, -1.5, 10.0, -1.5, 1.5, 10.0, -1.5
139   ]);
...
159   var vertices_finger = new Float32Array([ // Пальцы (1x2x1)
...
166   ]);
167 }
```



```

168 // Нормали
169 var normals = new Float32Array([
170     ...
171 ]);
172
173 // Индексы вершин
174 var indices = new Uint8Array([
175     0, 1, 2, 0, 2, 3, // передняя грань
176     4, 5, 6, 4, 6, 7, // правая грань
177     ...
178     20,21,22, 20,22,23 // задняя грань
179 ]);
180
181 // Переписать координаты в буферы, но не передавать в переменные-атрибуты
182 g_baseBuffer = initArrayBufferForLaterUse(gl, vertices_base, 3, gl.FLOAT);
183 g_arm1Buffer = initArrayBufferForLaterUse(gl, vertices_arm1, 3, gl.FLOAT);
184 ...
185 g_fingerBuffer = initArrayBufferForLaterUse(gl, vertices_finger, 3, gl.FLOAT);
186 ...
187 // Переписать нормали в буфер и передать его в переменную a_Normal
188 if (!initArrayBuffer(gl, 'a_Normal', normals, 3, gl.FLOAT)) return null;
189
190 // Переписать индексы в буфер
191 var indexBuffer = gl.createBuffer();
192 ...
193 gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
194 gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, indices, gl.STATIC_DRAW);
195
196 return indices.length;
197 }
198
199 ...
200 function draw(gl, n, viewProjMatrix, a_Position, u_MvpMatrix, u_NormalMatrix) {
201     ...
202     // Нарисовать станину
203     var baseHeight = 2.0;
204     g_modelMatrix.setTranslate(0.0, -12.0, 0.0);
205     drawSegment(gl, n, g_baseBuffer, viewProjMatrix, a_Position,
206                 u_MvpMatrix, u_NormalMatrix);
207
208     // Сегмент1
209     var arm1Length = 10.0;
210     g_modelMatrix.translate(0.0, baseHeight, 0.0); // Перенести на станину
211     g_modelMatrix.rotate(g_arm1Angle, 0.0, 1.0, 0.0); // Поворот (ось Y)
212     drawSegment(gl, n, g_arm1Buffer, viewProjMatrix, a_Position,
213                 u_MvpMatrix, u_NormalMatrix);
214
215     // Сегмент2
216     ...
217     // Палец2
218     ...
219     drawSegment(gl, n, g_fingerBuffer, viewProjMatrix, a_Position,
220                 u_MvpMatrix, u_NormalMatrix);
221 }
222
223 ...

```



```

310 // Рисует сегменты
311 function drawSegment(gl, n, buffer, viewProjMatrix, a_Position,
                      u_MvpMatrix, u_NormalMatrix) {
312     gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
313     // Присвоить буферный объект переменной-атрибуту
314     gl.vertexAttribPointer(a_Position, buffer.num, buffer.type, false, 0,
315 0);
316     // Разрешить присваивание
317     gl.enableVertexAttribArray(a_Position);
318     // Вычислить матрицу модели вида проекции и передать ее в u_MvpMatrix
319     ...
320 // Вычислить матрицу преобразования нормали и передать ее в u_NormalMatrix
321     ...
322     gl.drawElements(gl.TRIANGLES, n, gl.UNSIGNED_BYTE, 0);
323 }

```

Ключевыми особенностями этой программы являются: (1) наличие отдельных буферных объектов с координатами вершин для каждого сегмента, (2) перед рисованием каждого сегмента выполняется передача соответствующего буферного объекта в переменную `a_Position` и (3) разрешение присваивания и рисование сегмента.

Функция `main()`, определение которой начинается в строке 29, выполняет ту же последовательность операций, что и прежде. Инициализация буферов с координатами сегментов осуществляется в функции `initVertexBuffers()`, которая вызывается в строке 47. В строке 58 программа получает ссылку на переменную-атрибут `a_Position` и затем вызывает `draw()` (строка 73).

Рассмотрим функцию `initVertexBuffers()`, определение которой начинается в строке 121. В строках со 115 по 119 объявляются буферные объекты, используемые для хранения координат вершин всех сегментов. Основным отличием этой версии функции `initVertexBuffers()` от версии в программе `MultiJointModel.js` является определение координат (начинается со строки 123). Так как теперь каждый сегмент представлен собственным кубоидом, требуется определить координаты вершин всех сегментов в отдельности (например, для станины (`vertices_base`) в строке 123, для сегмента1 (`vertices_arm1`) в строке 132 и так далее). Фактическое создание буферных объектов для каждого элемента руки-манипулятора осуществляется функцией `initArrayBufferForLaterUse()`, в строках со 189 по 193. Эта функция представлена ниже:

```

211 function initArrayBufferForLaterUse(gl, data, num, type){
212     var buffer = gl.createBuffer(); // Создать буферный объект
213     ...
214     // Записать данные в буферный объект
215     gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
216     gl.bufferData(gl.ARRAY_BUFFER, data, gl.STATIC_DRAW);
217     // Сохранение информации в переменной-атрибуте выполняется позже
218     buffer.num = num;
219     buffer.type = type;

```

```

224
225     return buffer;
226 }

```

Функция `initArrayBufferForLaterUse()` просто создает буферный объект (строка 212) и записывает в него данные (строки 218 и 219). Примечательно, что сохранение информации в переменной-атрибуте (`gl.vertexAttribPointer()`) и разрешение присваивания (`gl.enableVertexAttribArray()`) выполняются за пределами этой функции, непосредственно перед рисованием. Чтобы позднее можно было присвоить буферный объект переменной-атрибуту `a_Position`, в свойствах буфера требуется сохранить дополнительные данные (строки 222 и 223).

Здесь мы воспользовались интересной особенностью JavaScript, которая позволяет добавлять произвольные свойства к любому объекту и сохранять в них данные. Для этого достаточно добавить к имени объекта `.имя_свойства` и присвоить значение. С помощью этой особенности мы сохраняем число элементов в свойстве `num` (строка 222) и тип в свойстве `type` (строка 223). Разумеется, доступ к значениям вновь созданных свойств осуществляется по тем же именам. Будьте внимательны при обращении к свойствам, созданным таким способом, потому что JavaScript не сообщает об ошибке, если в имени подобного свойства допустить опечатку. Имейте также в виду, что использование подобных свойств отрицательно сказывается на производительности. Более удачное решение заключается в использовании собственных типов, как описывается в главе 10, «Продвинутые приемы», но давайте пока будем использовать этот прием.

Наконец, в этой программе используется практически такая же функция `draw()`, как в программе `MultiJointModel`, в том смысле, что она рисует элементы в соответствии с иерархической структурой модели, но для рисования каждого сегмента она вызывает `drawSegment()`. В третьем аргументе функции `drawSegment()`, показанной ниже, передается буферный объект с координатами сегмента.

```

262     drawSegment(gl, n, g_baseBuffer, viewProjMatrix, u_MvpMatrix, u_NormalMatrix);

```

Определение этой функции начинается в строке 311, а действует она следующим образом: в строках с 312 по 316 буферный объект присваивается переменной-атрибуту `a_Position` и затем, в строке 327 осуществляется рисование. Здесь также используются свойства `num` и `type` буферного объекта, которые были созданы ранее.

```

310 // Рисует сегменты
311 function drawSegment(gl, n, buffer, viewProjMatrix, a_Position,
                      u_MvpMatrix, u_NormalMatrix) {
312     gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
313     // Присвоить буферный объект переменной-атрибуту
314     gl.vertexAttribPointer(a_Position, buffer.num, buffer.type, false, 0,
315                             0);
316     // Разрешить присваивание
317     gl.enableVertexAttribArray(a_Position);
318     // Вычислить матрицу модели вида проекции и передать ее в u_MvpMatrix

```

```
...
322 // Вычислить матрицу преобразования нормали и передать ее в u_NormalMatrix
...
327 gl.drawElements(gl.TRIANGLES, n, gl.UNSIGNED_BYTE, 0);
328 }
```

На этот раз нет необходимости масштабировать объекты с применением матрицы модели, потому что они подготовлены заранее и имеют нужные размеры, соответственно отпала необходимость сохранять и восстанавливать матрицу. По этой причине функции `pushMatrix()` и `popMatrix()` в этой программе стали не нужны.

Шейдер и объект программы: роль `initShaders()`

В заключение, прежде чем завершить эту главу, исследуем одну интересную функцию: `initShaders()`. Эта функция повсеместно используется в книге и скрывает за своим интерфейсом массу тонкостей, связанных с настройкой и использованием шейдеров. Мы преднамеренно отложили знакомство с ней до конца главы, чтобы вы могли уяснить основные особенности WebGL, прежде чем переходить к изучению сложных деталей. Следует заметить, что для успешной работы совсем не требуется знать и понимать все тонкости, описываемые ниже. Многим будет вполне достаточно просто использовать функцию `initShaders()`, которую мы написали для вас, и вы спокойно можете пропустить этот раздел. Однако, если вам интересно, продолжайте чтение.

Функция `initShaders()` делает всю рутинную работу по подготовке шейдеров к работе. Она выполняет следующие семь шагов:

1. Создает объекты шейдеров (`gl.createShader()`).
2. Сохраняет шейдеры (чтобы избежать путаницы, мы будем называть их как «исходный код») в объектах шейдеров (`g.shaderSource()`).
3. Компилирует объекты шейдеров (`gl.compileShader()`).
4. Создает объект программы (`gl.createProgram()`).
5. Подключает объекты шейдеров к объекту программы (`gl.attachShader()`).
6. Компонирует объект программы (`gl.linkProgram()`).
7. Сообщает системе WebGL, что объект программы готов к использованию (`gl.useProgram()`).

Каждый шаг сам по себе прост, но их комбинация кому-то из вас может показаться сложной, поэтому давайте рассмотрим их один за другим. Во-первых, как вы уже знаете, для использования шейдеров необходимы объекты двух типов: объекты шейдеров и объекты программ.

Объект шейдера

Объект шейдера управляет вершинным или фрагментным шейдером. Для каждого шейдера создается один объект.

Объект программы Объект программы – это контейнер, управляющий объектами шейдеров. Объект вершинного шейдера и объект фрагментного шейдера (всего два объекта) должны явно подключаться к объекту программы.

На рис. 9.10 показано, как взаимосвязаны между собой объект программы и объекты шейдеров.



Рис. 9.10. Взаимосвязь объекта программы и объектов шейдеров

Теперь, используя эту информацию, обсудим все семь шагов.

Создание объектов шейдеров (*gl.createShader()*)

Все объекты шейдеров создаются вызовом `gl.createShader()`.

gl.createShader (type)		
Создает объект шейдера указанного типа <code>type</code> .		
Параметры	<code>type</code>	Определяет тип создаваемого объекта шейдера. Может иметь одно из двух значений: <code>gl.VERTEX_SHADER</code> (для вершинного шейдера) или <code>gl.FRAGMENT_SHADER</code> (для фрагментного шейдера).
Возвращаемое значение	непустое значение	Ссылка на созданный объект шейдера.
	<code>null</code>	Попытка создать объект шейдера не увенчалась успехом.
Ошибки	<code>INVALID_ENUM</code>	Аргумент <code>type</code> имеет недопустимое значение.

Функция `gl.createShader()` создает вершинный или фрагментный шейдер, в зависимости от значения аргумента `type`. Когда шейдер станет ненужным, его можно удалить с помощью `gl.deleteShader()`.

gl.deleteShader (shader)		
Удаляет объект шейдера <code>shader</code> .		
Параметры	<code>shader</code>	Определяет объект шейдера, подлежащий удалению.
Возвращаемое значение	нет	
Ошибки	нет	

Имейте в виду, что указанный объект шейдера не будет удален немедленно, если он продолжает использоваться (то есть, если он подключен к объекту программы вызовом функции `gl.attachShader()`, о которой рассказывается несколькими страницами ниже). Объект шейдера, что передается функции `gl.deleteShader()` в виде аргумента, будет удален, когда бъект программы перестанет использовать его.

Сохранение исходного кода шейдеров в объектах шейдеров (`gl.shaderSource()`)

Объект шейдера имеет хранилище, где хранится исходный код (тот, что в программах на JavaScript определяется в виде строки или в виде отдельного файла; см. приложение F, «Загрузка шейдеров из файлов»). Сохранение исходного кода шейдера в объекте осуществляется вызовом функции `gl.shaderSource()`.

`gl.shaderSource(shader, source)`

Сохраняет исходный код `source` в объекте шейдера `shader`. Если прежде в объекте шейдера уже сохранялся исходный код, он будет замещен содержимым аргумента `source`.

Параметры	<code>shader</code>	Определяет объекта шейдера, в котором должен быть сохранен исходный код.
-----------	---------------------	--

	<code>source</code>	Определяет исходный код шейдера (строка).
--	---------------------	---

Возвращаемое значение	нет
-----------------------	-----

Ошибки	нет
--------	-----

Компиляция объектов шейдеров (`gl.compileShader()`)

После сохранения исходного кода в объекте шейдера его необходимо скомпилировать, чтобы система WebGL получила возможность использовать его. В отличие от программ на JavaScript и подобно программам на C или C++, шейдеры должны компилироваться перед применением. В данном случае исходный код, хранящийся в объекте шейдера, компилируется в выполняемый (двоичный) формат и передается системе WebGL. Компиляция осуществляется с помощью функции `gl.compileShader()`. Имейте в виду, если исходный код в объекте шейдера заменить вызовом функции `gl.shaderSource()` уже после компиляции, скомпилированный код останется прежним. Его необходимо скомпилировать повторно.

`gl.compileShader(shader)`

Компилирует исходный код, хранящийся в объекте шейдера `shader`.

Параметры	shader	Определяет объекта шейдера, в котором хранится исходный код, подлежащий компиляции.
Возвращаемое значение	нет	
Ошибки	нет	

В процессе выполнения `gl.compileShader()` могут возникать ошибки компиляции из-за ошибок в исходном коде. Проверить наличие ошибок компиляции, а также состояние объекта шейдера можно с помощью функции `gl.getShaderParameter()`.

`gl.getShaderParameter(shader, pname)`

Возвращает информацию, определяемую параметром `pname`, для объекта шейдера `shader`

Параметры	shader	Определяет объекта шейдера.
	pname	Определяет информацию, которую требуется вернуть: <code>gl.SHADER_TYPE</code> , <code>gl.DELETE_STATUS</code> или <code>gl.COMPILE_STATUS</code> .
Возвращаемое значение	в зависимости от <code>pname</code> :	
	<code>gl.SHADER_TYPE</code>	Тип шейдера (<code>gl.VERTEX_SHADER</code> или <code>gl.FRAGMENT_SHADER</code>)
	<code>gl.DELETE_STATUS</code>	Преуспела ли операция удаления шейдера (<code>true</code> или <code>false</code>)
	<code>gl.COMPILE_STATUS</code>	Преуспела ли операция компиляции шейдера (<code>true</code> или <code>false</code>)
Ошибки	<code>INVALID_ENUM</code>	В аргументе <code>pname</code> передано недопустимое значение.

Чтобы проверить успешность компиляции, можно вызвать `gl.getShaderParameter()` со значением `gl.COMPILE_STATUS` аргумента `pname`.

Если во время компиляции были обнаружены ошибки, `gl.getShaderParameter()` вернет `false` и запишет информацию об ошибке в журнал. Эту информацию можно извлечь вызовом `gl.getShaderInfoLog()`.

`gl.getShaderInfoLog(shader)`

Возвращает информацию из журнала для объекта шейдера `shader`.

Параметры	shader	Определяет объекта шейдера, для которого следует извлечь информацию из журнала.
Возвращаемое значение	непустое значение	Строка с информацией из журнала.
	null	Ошибок не обнаружено.
Ошибки	нет	

Несмотря на то, что точное содержимое информации, возвращаемой функцией `gl.getShaderInfoLog()`, зависит от реализации, практически все системы WebGL возвращают текст сообщений с номерами строк, где компилятор обнаружил ошибки. Например, допустим, что мы попытались скомпилировать следующий шейдер:

```
var FSHADER_SOURCE =
    'void main() {\n' +
    '    gl.FragColor = vec4(1.0, 0.0, 0.0, 1.0);\n' +
    '}\n';
```

Данный шейдер содержит ошибку во второй строке (вместо `gl.` должно быть `gl_`), поэтому в консоли JavaScript, в браузере Chrome, сообщение об ошибке будет иметь вид, как показано на рис. 9.11.

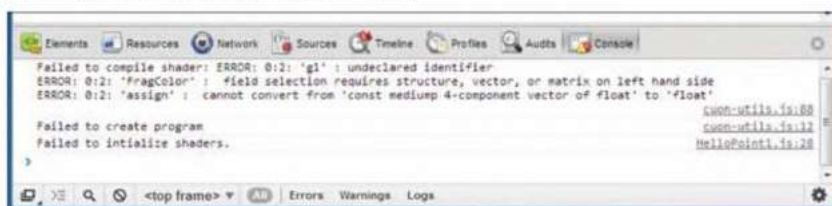


Рис. 9.11. Ошибка компиляции шейдера

В первом сообщении говорится, что объект `gl`, используемый в строке 2, не был объявлен.

```
failed to compile shader: ERROR: 0: 2 : 'gl' : undeclared identifier
                                cuon-utils.js:88
(ошибка компиляции шейдера: ОШИБКА: 0: 2 : 'gl' : необъявленный идентификатор
                                cuon-utils.js:88)
```

Ссылка на `cuon-utils.js:88` справа означает, что ошибка была обнаружена функцией `gl.getShaderInfoLog()` в строке 88, в файле `cuon-utils.js`, где определена функция `initShaders()`.

Создание объекта программы (`gl.createProgram()`)

Как уже упоминалось выше, объект программы является контейнером, где хранятся объекты шейдеров, и создается вызовом `gl.createProgram()`. Вы уже видели этот объект прежде – мы передавали его в первом аргументе функциям `gl.getAttribLocation()` и `gl.getUniformLocation()`.

`gl.createProgram()`

Создает объект программы.

Параметры нет

Возвращаемое значение	непустое значение	Созданный объект программы..
	null	Произошла ошибка при создании объекта программы.
Ошибки	нет	

Удалить объект программы можно с помощью `gl.deleteProgram()`.

`gl.deleteProgram(program)`

Удаляет объект программы `program`. Если в системе не осталось ссылок на объект программы, он удаляется немедленно. В противном случае он будет удален после исчезновения последней ссылки на него.

Параметры	<code>program</code>	Определяет объект программы, подлежащий удалению.
Возвращаемое значение	нет	
Ошибки	нет	

После создания объекта программы к нему следует подключить два объекта шейдеров.

Подключение объектов шейдеров к объекту программы (`gl.attachShader()`)

Так как в системе WebGL всегда требуется наличие двух шейдеров – вершинного и фрагментного – к объекту программы нужно подключить оба объекта. Делается это вызовом `gl.attachShader()`.

`gl.attachShader(program, shader)`

Подключает объект шейдера `shader` к объекту программы `program`.

Параметры	<code>program</code>	Определяет объект программы.
	<code>shader</code>	Определяет объекта шейдера, который требуется подключить.
Возвращаемое значение	нет	
Ошибки	<code>INVALID_OPERATION</code>	Объект шейдера <code>shader</code> уже подключен к объекту программы <code>program</code> .

Совсем необязательно сохранять исходный код в объекте шейдера или компилировать этот код перед подключением к объекту программы. Отключить объект шейдера можно с помощью `gl.detachShader()`.

`gl.detachShader(program, shader)`

Отключает объект шейдера `shader` от объекта программы `program`.

Параметры	<code>program</code>	Определяет объект программы.
	<code>shader</code>	Определяет объекта шейдера, который требуется отключить.
Возвращаемое значение	нет	
Ошибки	<code>INVALID_OPERATION</code>	Объект шейдера <code>shader</code> не подключен к объекту программы <code>program</code> .

Компоновка объекта программы (`gl.linkProgram()`)

После подключения объекта шейдера к объекту программы необходимо скомпоновать объекты шейдеров с объектом программы. Сделать это можно с помощью функции `gl.linkProgram()`.

<code>gl.linkProgram(program)</code>		
Выполняет компоновку объекта программы <code>program</code> .		
Параметры	<code>program</code>	Определяет объект программы, подлежащий компоновке.
Возвращаемое значение	нет	
Ошибки	нет	

В процессе компоновки проверяются различные ограничения системы WebGL: (1) при наличии объявлений `varying`-переменных в вершинном шейдере проверяется объявление одноименных `varying`-переменных того же типа во фрагментном шейдере, (2) проверяется наличие в вершинном шейдере инструкций записи данных в `varying`-переменные, используемые во фрагментном шейдере, (3) проверяется совпадение типов одноименных `uniform`-переменных, используемых в вершинном и фрагментном шейдерах, (4) проверяется, не превышает ли число переменных-атрибутов, `uniform`-переменных и `varying`-переменных установленных пределов, и так далее.

После компоновки объекта программы никогда нелишним будет проверить ее успешность. Сделать это можно с помощью функции `gl.getProgramParameters()`.

<code>gl.getProgramParameter(program, pname)</code>		
Возвращает информацию, определяемую параметром <code>pname</code> , для объекта программы <code>program</code> . Возвращаемое значение зависит от значения параметра <code>pname</code> .		
Параметры	<code>program</code>	Определяет объекта программы.
	<code>pname</code>	Определяет информацию, которую требуется вернуть: <code>gl.DELETE_STATUS</code> , <code>gl.LINK_STATUS</code> , <code>gl.VALIDATE_STATUS</code> , <code>gl.ATTACHED_SHADERS</code> , <code>gl.ACTIVE_ATTRIBUTES</code> или <code>gl.ACTIVE_UNIFORMS</code> .

Возвращаемое значение	в зависимости от pname:	
	<code>gl.DELETE_STATUS</code>	Был ли удален объект программы (true или false)
	<code>gl.LINK_STATUS</code>	Преуспела ли операция компоновки программы (true или false)
	<code>gl.VALIDATE_STATUS</code>	Преуспела ли операция проверки объекта программы (true или false) ¹
	<code>gl.ATTACHED_SHADERS</code>	Число подключенных объектов шейдеров
	<code>gl.ACTIVE_ATTRIBUTES</code>	Число переменных-атрибутов в вершинном шейдере
	<code>gl.ACTIVE_UNIFORMS</code>	Число uniform-переменных
Ошибки	<code>INVALID_ENUM</code>	В аргументе pname передано недопустимое значение.

¹ Объект программы может генерировать ошибки во время выполнения, даже если компоновка прошла успешно, например, из-за того, что не был инициализирован текстурный слот. Подобные ошибки проявляются только во время рисования, но не во время компоновки. Поскольку данная проверка влияет на производительность, выполнять ее желательно только на этапе разработки и отладки.

Если компоновка пройдет успешно, `gl.linkProgram()` вернет объект выполняемой программы. В случае ошибки информацию о ней можно получить вызовом `gl.getProgramInfoLog()`.

`gl.getProgramInfoLog(program)`

Возвращает информацию из журнала для объекта программы `program`.

Параметры `program` Определяет объекта программы, для которого следует извлечь информацию из журнала.

Возвращаемое значение Строка с информацией из журнала.

Ошибки нет

Сообщение системе WebGL о готовности объекта программы (`gl.useProgram()`)

Последний шаг – сообщение системе WebGL о готовности объекта программы к использованию вызовом `gl.useProgram()`.

`gl.useProgram(program)`

Сообщает системе WebGL, что объект программы `program` готов к использованию.

Параметры `program` Определяет объекта программы, который следует использовать.

Возвращаемое значение нет

Ошибки нет

Одной из мощных особенностей этой функции является возможность переключения между объектами программ с шейдерами, подготовленными заранее, прямо во время рисования. Она детально будет обсуждаться и демонстрироваться в главе 10.

Этим последним шагом завершается подготовка шейдеров к рисованию. Как видите, функция `initShaders()` скрывает массу тонкостей за своим интерфейсом, и вы можете пользоваться ею, не вникая в особенности ее работы. Фактически, после вызова этой функции вершинный и фрагментный шейдеры будут полностью готовы к работе, и вы можете просто пользоваться ими, вызывая функции `gl.drawArrays()` и `gl.drawElements()`.

Теперь, когда вы получили представление о том, какие шаги выполняет и какие функции вызывает `initShaders()`, рассмотрим фактическую ее реализацию в библиотеке `cuon-utils.js`.

Реализация `initShaders()`

Функция `initShaders()` опирается на две основные функции: `createProgram()`, которая создает скомпонованный объект программы, и `loadShader()`, которая вызывается из `createProgram()` и создает скомпилированные объекты шейдеров. Обе функции определены в библиотеке `cuon-utils.js`. Далее мы рассмотрим реализацию `initShader()` сверху вниз (см. листинг 9.5). Обратите внимание, что в отличие от обычных примеров, которые приводятся в этой книге, комментарии в библиотеке `cuon-utils.js` для большего удобства написаны в формате `JavaDoc`.

Листинг 9.5. `initShaders()`

```
1 // cuon-utils.js
2 /**
3  * Создает объект программы и делает его текущим
4  * @param gl контекст GL
5  * @param vshader вершинный шейдер (строка)
6  * @param fshader фрагментный шейдер (строка)
7  * @return true, если объект программы успешно создан и сделан текущим
8  */
9 function initShaders(gl, vshader, fshader) {
10   var program = createProgram(gl, vshader, fshader);
11   ...
16   gl.useProgram(program);
17   gl.program = program;
18
19   return true;
20 }
```

В первую очередь функция `initShaders()` создает скомпонованный объект программы вызовом `createProgram()` в строке 10 и затем, в строке 16, сообщает системе WebGL, что он готов к использованию. После этого ссылка на объект программы сохраняется в свойстве с именем `program` объекта `gl`.

Теперь перейдем к функции `createProgram()` (листинг 9.6).

Листинг 9.6. `createProgram()`

```
22 /**
23  * Создает скомпонованный объект программы
24  * @param gl        контекст GL
25  * @param vshader   вершинный шейдер (строка)
26  * @param fshader   фрагментный шейдер (строка)
27  * @return          созданный объект программы или null, если возникла ошибка.
28  */
29 function createProgram(gl, vshader, fshader) {
30     // Создать объекты шейдеров
31     var vertexShader = loadShader(gl, gl.VERTEX_SHADER, vshader);
32     var fragmentShader = loadShader(gl, gl.FRAGMENT_SHADER, fshader);
33     ...
34     // Создать объект программы
35     var program = gl.createProgram();
36     ...
37     // Подключить объекты шейдеров
38     gl.attachShader(program, vertexShader);
39     gl.attachShader(program, fragmentShader);
40     ...
41     // Скомпоновать объект программы
42     gl.linkProgram(program);
43     ...
44     // Проверить результат компоновки
45     var linked = gl.getProgramParameter(program, gl.LINK_STATUS);
46     ...
47     return program;
48 }
49 }
```

Функция `createProgram()` создает объекты вершинного и фрагментного шейдеров, которые загружаются вызовами функции `loadShader()` в строках 31 и 32. Объект шейдера, возвращаемый функцией `loadShader()`, уже хранит исходный код и скомпилированную его версию.

В строке 38 создается объект программы, после чего, в строках 44 и 45, к нему подключаются только что созданные объекты вершинного и фрагментного шейдеров. Затем `createProgram()` компоует объект программы в строке 48 и в строке 51 проверяет результат. Если компоновка прошла успешно, она возвращает готовый объект программы (строка 60).

В заключение рассмотрим функцию `loadShader()` (листинг 9.7), которая вызывается в строках 31 и 32 из функции `createProgram()`.

Листинг 9.7. `loadShader()`

```
63 /**
64  * Создает объект шейдера
65  * @param gl        контекст GL
66  * @param type      тип создаваемого объекта шейдера
67  * @param source     исходный код шейдера (строка)
```



```
68 * @return          созданный объект шейдера или null в случае ошибки.
69 */
70 function loadShader(gl, type, source) {
71     // Создать объект шейдера
72     var shader = gl.createShader(type);
73     ...
74     // Сохранить исходный код шейдера
75     gl.shaderSource(shader, source);
76
77     // Скомпилировать шейдер
78     gl.compileShader(shader);
79
80     // Проверить результат компиляции
81     var compiled = gl.getShaderParameter(shader, gl.COMPILE_STATUS);
82     ...
83     return shader;
84 }
```

Прежде всего `loadShader()` создает объект шейдера в строке 72. В строке 79 в объекте сохраняется исходный код шейдера и в строке 82 выполняется его компиляция. В завершение проверяется результат компиляции (строка 85) и, если компиляция прошла успешно, объект шейдера возвращается функции `createShader()`, которая подключит его к объекту программы.

В заключение

Эта глава завершает исследование базовых возможностей системы WebGL. Здесь вы узнали, как рисовать сложные трехмерные объекты, состоящие из нескольких сегментов, и как управлять ими. Описанные приемы помогают лучше понять, как использовать простые трехмерные объекты, такие как кубы и блоки, для конструирования более сложных объектов. Кроме того, здесь вы познакомились с одной из наиболее сложных вспомогательных функций, написанных специально для этой книги — `initShaders()` — которая до этой главы использовалась нами как «черный ящик». Вы узнали, как создаются объекты шейдеров и как они управляются объектами программ, и теперь имеете более полное представление о внутреннем устройстве шейдеров, и как система WebGL управляет ими посредством объектов программ.

На данном этапе вы обладаете всеми необходимыми знаниями, чтобы начать писать свои трехмерные сцены, используя возможности системы WebGL. В следующей главе мы расскажем в общих чертах о различных продвинутых приемах работы с трехмерной графикой и покажем, как они поддерживаются системой WebGL.

ГЛАВА 10.

Продвинутые приемы

Эта глава включает целый «мешок» интересных приемов, которые наверняка пригодятся вам при создании своих WebGL-приложений. Основную массу составляют отдельные, обособленные приемы, поэтому вы свободно можете перепрыгивать между разделами и читать те, которые будут вам интересны. Приемы, имеющие зависимости, мы будем обозначать явно. Описания в этой главе будут краткими, чтобы уместить в нее как можно больше приемов. Зато примеры программ, которые можно найти на сайте книги, содержат вполне исчерпывающие комментарии – помните об этом и читайте их.

Вращение объекта мышью

В WebGL-приложениях иногда бывает желательно дать пользователю возможность управлять трехмерными объектами мышью. В этом разделе мы сконструируем программу `RotateObject`, которая позволяет вращать куб мышью. Чтобы упростить программу, мы будем использовать куб, но описываемый здесь метод можно применить к любому объекту. На рис. 10.1 показано, как выглядит куб с наложенной на него текстурой, который рисует программа.

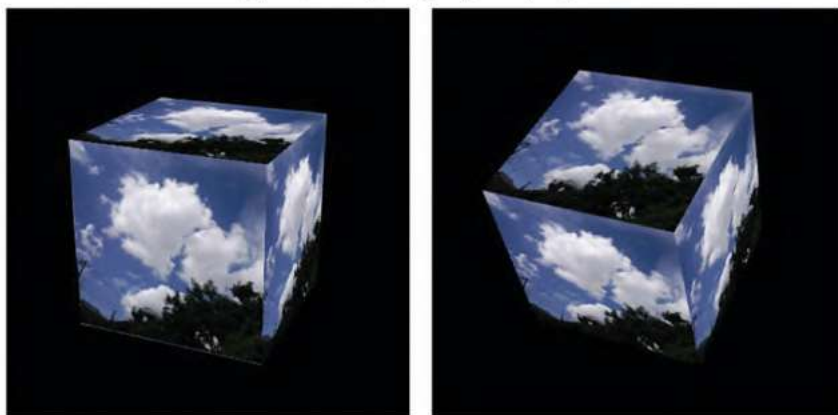


Рис. 10.1. Скриншот программы `RotateObject`

Как реализовать вращение объекта

Вращение трехмерного объекта является разновидностью приема преобразования координат вершин с использованием матрицы модели вида проекции, который мы уже использовали для управления 2-мерными объектами. Для этого требуется создать матрицу вращения в ответ на перемещение указателя мыши, изменить ее и затем выполнить преобразование координат с помощью этой матрицы.

Определить движение указателя мыши можно простым сохранением координат начального щелчка и последующим отслеживанием координат указателя. Очевидно, что для этих целей необходим обработчик событий, который будет преобразовывать перемещения указателя в угол поворота объекта. Рассмотрим, как реализовать это на практике.

Пример программы (RotateObject.js)

В листинге 10.1 приводится пример программы. Как можно заметить, шейдеры не содержат ничего особенного. Строка 9 в вершинном шейдере преобразует координаты вершины с использованием матрицы модели вида проекции, а строка 10 накладывает текстуру на куб.

Листинг 10.1. RotateObject.js

```
1 // RotateObject.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4     ...
5     'void main() {\n' +
6     '    gl_Position = u_MvpMatrix * a_Position;\n' +
7     '    v_TexCoord = a_TexCoord;\n' +
8     '}\n';
9
10 ...
11 function main() {
12     ...
13     var n = initVertexBuffers(gl);
14     ...
15     viewProjMatrix.setPerspective(30.0, canvas.width / canvas.height,
16                                   1.0, 100.0);
17     viewProjMatrix.lookAt(3.0, 3.0, 7.0, 0.0, 0.0, 0.0, 1.0, 0.0);
18     // Зарегистрировать обработчик событий
19     var currentAngle = [0.0, 0.0]; // [ось X, ось Y] градусы
20     initEventHandlers(canvas, currentAngle);
21     ...
22     var tick = function() { // Начало рисования
23         draw(gl, n, viewProjMatrix, u_MvpMatrix, currentAngle);
24         requestAnimationFrame(tick, canvas);
25     };
26     tick();
27 }
28
29 ...
30 function initEventHandlers(canvas, currentAngle) {
```

```
139 var dragging = false; // Буксировать или нет
140 var lastX = -1, lastY = -1; // Последняя позиция указателя мыши
141
142 canvas.onmousedown = function(ev) { // Кнопка мыши нажата
143     var x = ev.clientX, y = ev.clientY;
144     // Начать буксировку, если указатель в пределах элемента <canvas>
145     var rect = ev.target.getBoundingClientRect();
146     if (rect.left <= x && x < rect.right && rect.top <= y && y < rect.bottom) {
147         lastX = x; lastY = y;
148         dragging = true;
149     }
150 };
151 // Кнопка мыши отпущена
152 canvas.onmouseup = function(ev) { dragging = false; };
153
154 canvas.onmousemove = function(ev) { // Перемещение указателя
155     var x = ev.clientX, y = ev.clientY;
156     if (dragging) {
157         var factor = 100/canvas.height; // Скорость вращения
158         var dx = factor * (x - lastX);
159         var dy = factor * (y - lastY);
160         // Ограничить угол поворота по оси X от -90 до 90 градусов
161         currentAngle[0] = Math.max(Math.min(currentAngle[0] + dy, 90.0), -90.0);
162         currentAngle[1] = currentAngle[1] + dx;
163     }
164     lastX = x, lastY = y;
165 };
166 }
167
168 var g_MvpMatrix = new Matrix4(); // Матрица модели вида проекции
169 function draw(gl, n, viewProjMatrix, u_MvpMatrix, currentAngle) {
170     // Вычислить матрицу модели вида проекции
171     g_MvpMatrix.set(viewProjMatrix);
172     g_MvpMatrix.rotate(currentAngle[0], 1.0, 0.0, 0.0); // ось X
173     g_MvpMatrix.rotate(currentAngle[1], 0.0, 1.0, 0.0); // ось Y
174     gl.uniformMatrix4fv(u_MvpMatrix, false, g_MvpMatrix.elements);
175
176     gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
177     gl.drawElements(gl.TRIANGLES, n, gl.UNSIGNED_BYTE, 0);
178 }
```

В строках 61 и 62, внутри JavaScript-функции `main()`, заранее вычисляется матрица вида проекции. Матрицу модели необходимо будет вычислять «на лету», в зависимости от величины перемещения указателя мыши.

Код, начиная со строки 65, регистрирует обработчики событий, являющиеся ключевой частью этой программы. В строке 65 инициализируется переменная `currentAngle`, она будет использоваться для хранения текущего угла поворота. В данном случае она представляет собой массив, потому что программа должна обеспечивать вращение в двух плоскостях – относительно оси X и относительно оси Y. Фактическая регистрация обработчиков реализована внутри функции `initEventHandlers()`, которая вызывается в строке 66. Она рисует куб с помощью функции `tick()` (строка 74).

Определение функции `initEventHandlers()` начинается в строке 138. Код, начиная со строки 142, обрабатывает нажатие кнопки мыши, код, начиная со строки 152, обрабатывает отпускание кнопки мыши, а код, начиная со строки 154, обрабатывает перемещение указателя мыши.

Обработка нажатия кнопки мыши очень проста. В строке 146 проверяется, находится ли указатель мыши в пределах элемента `<canvas>`, и если это так, в строке 147 текущие координаты указателя сохраняются в переменных `lastX` и `lastY`. Затем, в строке 148, переменной `dragging` присваивается значение `true`, означающее, что буксировка мышью началась.

Обработка отпускания кнопки мыши, которая начинается в строке 152, так же не содержит ничего сложного. Так как отпускание кнопки означает конец буксировки мышью, этот обработчик просто присваивает переменной `dragging` значение `false`.

Обработчик, определение которого начинается в строке 154, является наиболее важным – он следит за перемещением указателя мыши. В строке 156 проверяется, сместился ли указатель, в строках 158 и 159 вычисляются расстояния перемещения по осям X и Y , и результаты сохраняются в переменных `dx` и `dy`. Эти значения масштабируются умножением на значение переменной `factor`, которое является функцией от размеров элемента `<canvas>`. Полученные расстояния используются затем для определения нового угла поворота прямым сложением расстояний с текущими углами в строках 161 и 162. При этом код ограничивает угол поворота диапазоном от -90 до $+90$ градусов, только чтобы показать такую возможность; вы можете просто удалить это ограничение. После этого текущая позиция указателя мыши сохраняется в переменных `lastX` и `lastY`.

После благополучного преобразования расстояний перемещения указателя мыши в углы поворота, на их основе, функция `tick()` вычислит матрицу поворота и нарисует повернутый куб. Эти операции выполняются в строках 172 и 173.

Представленный здесь прием вычисления угла поворота является лишь одним из многих. Описание других приемов, включая определение траектории движения объекта, можно найти в книге «3D User Interfaces».

Выбор объекта

Когда от приложения требуется, чтобы оно давал пользователю возможность управлять трехмерными объектами интерактивно, в нем часто бывает необходимо реализовать некоторый способ выбора объектов пользователем. Этот прием имеет массу применений, таких как выбор трехмерной кнопки, созданной внутри трехмерной модели, или выбор фотографии, находящейся среди множества других фотографий внутри трехмерной сцены.

Реализация выбора трехмерного объекта в общем случае является задачей более сложной, чем выбор 2-мерного объекта, потому что требуется привлекать сложный математический аппарат для определения фигуры, над которой находится указатель мыши. Однако, чтобы избежать лишних сложностей, вы може-

те использовать довольно простой трюк, показанный в примере программы. В этой программе, `PickObject`, пользователь может щелкнуть на вращающемся кубе и программа в ответ выведет диалог, как показано на рис. 10.2. Сначала запустите эту программу и поэкспериментируйте с ней, чтобы понять, как она работает.

На рис. 10.2 показано окно диалога, появляющееся после щелчка на кубе. Диалог содержит сообщение «The cube was selected!» («Был выбран куб!»). Проверьте также, что случится, если щелкнуть на черном пространстве фона.

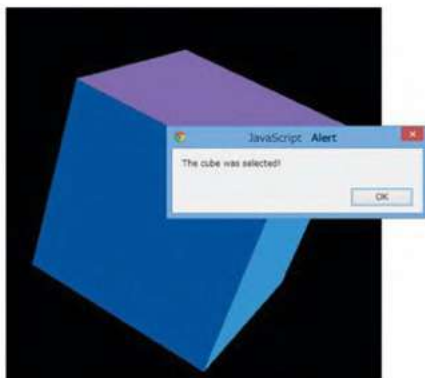


Рис. 10.2. `PickObject`

Как реализовать выбор объекта

Чтобы убедиться, что щелчок произошел на кубе, эта программа выполняет следующие шаги:

1. Когда нажимается кнопка мыши, куб перерисовывается одним красным цветом (см. рис. 10.3 в середине).
2. В точке указателя мыши определяется значение (цвет) пикселя.
3. Куб перерисовывается первоначальным цветом (на рис. 10.3 справа).
4. Если пиксель под указателем имел красный цвет, выводится диалог с сообщением «The cube was selected!».

Когда куб рисуется одним цветом (красным, в данном случае), легко определить область, которую он занимает. После чтения значения пикселя в позиции указателя мыши в момент щелчка, вы с уверенностью можете сказать, находился ли указатель над кубом.

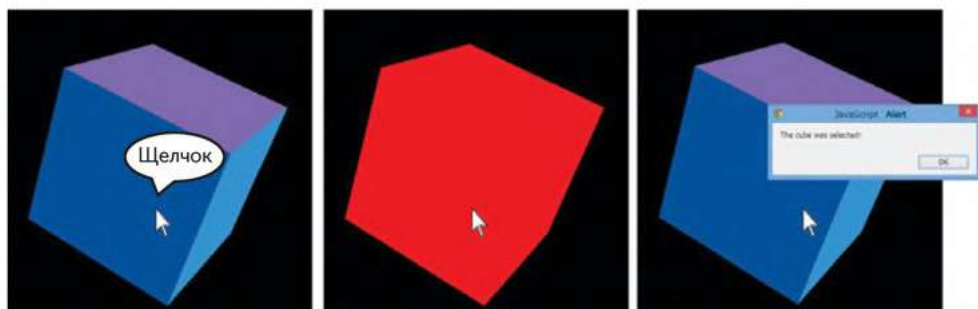


Рис. 10.3. Последовательность рисования объекта в момент щелчка мышью

Чтобы гарантировать, что пользователь не увидит куб, перекрашенный в красный цвет, его рисование и перерисовывание нужно выполнить в одной и той же функции. Перейдем теперь к фактической реализации примера программы.

Пример программы (PickObject.js)

Исходный код программы приводится в листинге 10.2. Основная работа здесь выполняется в вершинном шейдере. Чтобы реализовать шаг 1, необходимо сообщить вершинному шейдеру о нажатии кнопки мыши, чтобы он нарисовал куб красным цветом. Эта информация передается с помощью переменной `u_Clicked`, которая объявлена в строке 7, внутри вершинного шейдера. Когда пользователь нажимает кнопку мыши, переменной `u_Clicked` присваивается значение `true` в коде на JavaScript, которое затем проверяется в строке 11. Если переменная имеет значение `true`, переменной `v_Color` присваивается значение красного цвета; если нет – переменной `v_Color` присваивается значение `a_Color`. Благодаря этому при нажатии кнопки мыши куб перерисовывается красным цветом.

Листинг 10.2. PickObject.js

```

1 // PickObject.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4     ...
5     'uniform mat4 u_MvpMatrix;\n' +
6     'uniform bool u_Clicked;\n' + // Признак нажатия кнопки мыши
7     'varying vec4 v_Color;\n' +
8     'void main() {\n' +
9     '    gl_Position = u_MvpMatrix * a_Position;\n' +
10    '    if (u_Clicked) {\n' + // Если кнопка нажата, нарисовать красным <-(1)
11    '        v_Color = vec4(1.0, 0.0, 0.0, 1.0);\n' +
12    '    } else {\n' +
13    '        v_Color = a_Color;\n' +
14    '    }\n' +
15    '}\n';
16
17
18 // Фрагментный шейдер
19 ...
20
21 gl_FragColor = v_Color;\n' +
22
23 ...
24
25 function main() {
26     ...
27     var u_Clicked = gl.getUniformLocation(gl.program, 'u_Clicked');
28     ...
29     gl.uniform1i(u_Clicked, 0); // Передать false в u_Clicked
30
31     var currentAngle = 0.0; // Текущий угол поворота
32     // Зарегистрировать обработчик событий
33     canvas.onmousedown = function(ev) { // Кнопка нажата
34         var x = ev.clientX, y = ev.clientY;
35         var rect = ev.target.getBoundingClientRect();
36         if (rect.left <= x && x < rect.right && rect.top <= y && y < rect.bottom){
37             // Проверить - над объектом ли указатель мыши
38             var x_in_canvas = x - rect.left, y_in_canvas = rect.bottom - y;
39             var picked = check(gl, n, x_in_canvas, y_in_canvas, currentAngle,
40                               u_Clicked, viewProjMatrix, u_MvpMatrix);
41             if (picked) alert('The cube was selected! '); <-(4)
42         }
43     }
44 }

```



```

83     }
84 }
...
92 }
...
147 function check(gl, n, x, y, currentAngle, u_Clicked, viewProjMatrix,
    u_MvpMatrix) {
148     var picked = false;
149     gl.uniform1i(u_Clicked, 1); // Нарисовать куб красным
150     draw(gl, n, currentAngle, viewProjMatrix, u_MvpMatrix);
151     // Прочитать значение пикселя под указателем мыши
152     var pixels = new Uint8Array(4); // Массив для хранения пикселей
153     gl.readPixels(x, y, 1, 1, gl.RGBA, gl.UNSIGNED_BYTE, pixels);    <-(2)
154
155     if (pixels[0] == 255) // Указатель над кубом, если pixels[0] == 255
156         picked = true;
157
158     gl.uniform1i(u_Clicked, 0); // Передать false в u_Clicked: перерисовать
159     draw(gl, n, currentAngle, viewProjMatrix, u_MvpMatrix); //    <-(3)
160
161     return picked;
162 }

```

Начнем обсуждение со строки 30 – с определения JavaScript-функции `main()`. В строке 60 она получает ссылку на переменную `u_Clicked` и в строке 71 присваивает ей начальное значение `false`.

В строке 75 программа регистрирует обработчик событий, который должен вызываться по щелчку кнопкой мыши. Эта функция сначала проверяет, находится ли указатель мыши в пределах элемента `<canvas>` (строка 78) и если проверка прошла успешно, вызывает функцию `check()` в строке 81. Эта функция проверяет, принадлежат ли координаты, указанные в третьем и четвертом аргументах, области куба, (см. следующий абзац). Если проверка увенчалась успехом, она возвращает `true`, что вызывает вывод сообщения в строке 82.

Определение функции `check()` начинается в строке 147. Эта функция реализует шаги (2) и (3), описанные в предыдущем разделе. В строке 149 она сообщает вершинному шейдеру, что имело место события щелчка записью значения 1 (`true`) в переменную `u_Clicked`. Далее, в строке 150, она рисует куб, повернутый на величину текущего угла. Так как `u_Clicked` имеет значение `true`, куб рисуется красным цветом. Затем, в строке 153, из буфера цвета читается значение пикселя в позиции щелчка 153. Ниже приводится описание функции `gl.readPixels()`, используемой здесь.

`gl.readPixels(x, y, width, height, format, type, pixels)`

Читает блок пикселей из буфера цвета¹ и сохраняет их в массиве `pixels`. Параметры `x`, `y`, `width` и `height` определяют блок как область прямоугольной формы.

Параметры

`x, y`

Определяют координаты первого пикселя в блоке для чтения из буфера.

	<code>width, height</code>	Определяют размеры прямоугольной области в пикселях
	<code>format</code>	Определяет формат представления пикселей. Должен иметь значение <code>gl.RGBA</code> .
	<code>type</code>	Определяет тип данных, представляющих значения пикселей. Должен иметь значение <code>gl.UNSIGNED_BYTE</code> .
	<code>pixels</code>	Определяет типизированный массив (<code>Uint8Array</code>) для хранения значений пикселей.
Возвращаемое значение	нет	
Ошибки	<code>INVALID_VALUE</code>	В аргументе <code>pixels</code> передано значение <code>null</code> . Или один из аргументов, <code>width</code> или <code>height</code> , имеет отрицательное значение.
	<code>INVALID_OPERATION</code>	В массиве <code>pixels</code> недостаточно места для хранения данных.
	<code>INVALID_ENUM</code>	В аргументе <code>format</code> передано недопустимое значение.

¹ Если с типом `gl.FRAMEBUFFER` связать объект буфера кадра, этот метод прочитает значения из буфера кадра. Мы поговорим об этом объекте в разделе «Использование нарисованного изображения в качестве текстуры».

Значение пикселя сохраняется в массиве `pixels`. Этот массив создается в строке 152 и хранит значения компонентов цвета R, G, B и A в элементах `pixels[0]`, `pixels[1]`, `pixels[2]` и `pixels[3]`, соответственно. Так как в этой программе известно, что может быть только два цвета – красный для куба и черный для фона – легко выяснить, находился ли указатель мыши над кубом в момент щелчка, для чего достаточно проверить только значение `pixels[0]`. Эта проверка выполняется в строке 155, и если она увенчалась успехом, переменной `picked` присваивается значение `true`.

Затем, в строке 158, в переменную `u_Clicked` передается значение `false` и в строке 159 куб перерисовывается. Эта операция возвращает кубу первоначальный цвет. Строка 161 возвращает значение `picked` вызывающему коду.

Обратите внимание, что если в этот момент вызвать какую-нибудь функцию, возвращающую управление браузеру, такую как `alert()`, содержимое буфера цвета автоматически будет выведено в элемент `<canvas>`. Например, если в строке 156 вызвать `alert('The cube was selected!')`, в момент щелчка на экране появится красный куб.

Несмотря на его простоту, данный подход можно использовать для обработки множества объектов, присваивая им разные цвета. Например, для обработки трех объектов достаточно будет красного, синего и зеленого. Для большего числа объектов можно использовать отдельные биты. Так как в формате RGBA каждый компонент цвета представлен 8-битными значениями, мы можем различать 255 объектов, используя только компонент R. Однако, при наличии сложных трехмерных объектов или очень большой области рисования, процедура определения вы-

бранного объекта может занимать существенное время. Устранить этот недостаток можно применением упрощенных моделей выбираемых объектов или уменьшением области рисования. В таких ситуациях можно также воспользоваться буфером кадра, о котором подробнее рассказывается в разделе «Использование нарисованного изображения в качестве текстуры», далее в этой главе.

Выбор грани объекта

Метод, описанный выше, можно также задействовать для определения выбора той или иной грани объекта. Давайте переделаем программу `PickObject` в программу `PickFace`, которая будет окрашивать выбранную грань в белый цвет. На рис. 10.4 приводится скриншот программы `PickFace`.

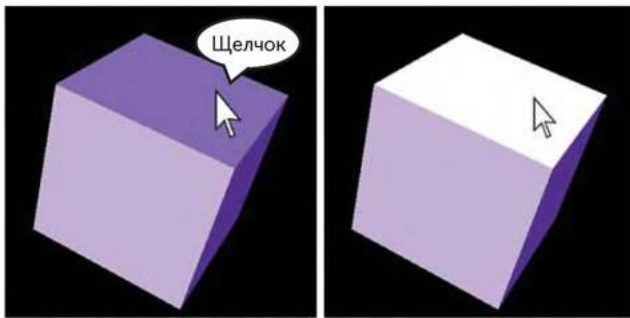


Рис. 10.4. `PickFace`

Вы легко поймете, как действует программа `PickFace`, если разобрались с тем, как работает `PickObject`. Программа `PickObject` окрашивает куб в красный цвет в момент щелчка мышью. После этого, читая значение пикселя в позиции щелчка, не составляет никакого труда узнать, был ли этот щелчок произведен в области куба. Программа `PickFace` идет дальше и определяет, на какой из граней куба был выполнен щелчок. Здесь мы будем хранить дополнительную информацию в альфа-компоненте значения цвета в формате RGBA. Рассмотрим исходный код программы.

Пример программы (`PickFace.js`)

Исходный код программы `PickFace.js` приводится в листинге 10.3. Некоторые части программы, такие как фрагментный шейдер, мы опустили ради экономии места.

Листинг 10.3. `PickFace.js`

```
1 // PickFace.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4     'attribute vec4 a_Position;\n' +
```

```

5   'attribute vec4 a_Color;\n' +
6   'attribute float a_Face;\n' + // Номер грани (нельзя использовать int)
7   'uniform mat4 u_MvpMatrix;\n' +
8   'uniform int u_PickedFace;\n' + // Номер выбранной грани
9   'varying vec4 v_Color;\n' +
10  'void main() {\n' +
11  '    gl_Position = u_MvpMatrix * a_Position;\n' +
12  '    int face = int(a_Face);\n' + // привести к типу int
13  '    vec3 color = (face == u_PickedFace) ? vec3(1.0):a_Color.rgb;\n'+
14  '    if(u_PickedFace == 0) {\n' + // Вставить номер грани в альфа-компонент
15  '        v_Color = vec4(color, a_Face/255.0);\n' +
16  '    } else {\n' +
17  '        v_Color = vec4(color, a_Color.a);\n' +
18  '    }\n' +
19  '}\n';

...
33 function main() {
    ...
50    // Определить координаты и цвет вершин
51    var n = initVertexBuffers(gl);
    ...
74    // Инициализировать выбранную грань
75    gl.uniform1i(u_PickedFace, -1);
76
77    var currentAngle = 0.0; // Текущий угол поворота (в градусах)
78    // Зарегистрировать обработчики событий
79    canvas.onmousedown = function(ev) { // Кнопка мыши нажата
80        var x = ev.clientX, y = ev.clientY;
81        var rect = ev.target.getBoundingClientRect();
82        if (rect.left <= x && x < rect.right && rect.top <= y && y < rect.bottom){
83            // Если позиция щелчка внутри <canvas>, обновить грань
84            var x_in_canvas = x - rect.left, y_in_canvas = rect.bottom - y;
85            var face = checkFace(gl, n, x_in_canvas, y_in_canvas,
86                currentAngle, u_PickedFace, viewProjMatrix, u_MvpMatrix);
87            gl.uniform1i(u_PickedFace, face); // Передать номер грани
88            draw(gl, n, currentAngle, viewProjMatrix, u_MvpMatrix);
89        }
    ...
99 function initVertexBuffers(gl) {
    ...
109    var vertices = new Float32Array([ // Координаты вершин
110        1.0, 1.0, 1.0, -1.0, 1.0, 1.0, -1.0,-1.0, 1.0, 1.0,-1.0, 1.0,
111        1.0, 1.0, 1.0, 1.0,-1.0, 1.0, 1.0,-1.0,-1.0, 1.0, 1.0,-1.0,
    ...
115        1.0,-1.0,-1.0, -1.0,-1.0,-1.0, -1.0, 1.0,-1.0, 1.0, 1.0,-1.0
116    ]);
    ...
127    var faces = new Uint8Array([ // Номера граней
128        1, 1, 1, 1, // v0-v1-v2-v3 Передняя
129        2, 2, 2, 2, // v0-v3-v4-v5 Правая
    ...
133        6, 6, 6, 6, // v4-v7-v6-v5 Задняя
134    ]);

```



```
...
154 if (!initArrayBuffer(gl, faces, gl.UNSIGNED_BYTE, 1,
                        'a_Face')) return -1; // Информация о грани
...
164 }
165
166 function checkFace(gl, n, x, y, currentAngle, u_PickedFace, viewProjMatrix,
                    u_MvpMatrix) {
167     var pixels = new Uint8Array(4); // Массив для хранения пикселей
168     gl.uniform1i(u_PickedFace, 0); // Записать номер грани в альфа-компонент
169     draw(gl, n, currentAngle, viewProjMatrix, u_MvpMatrix);
170     // Прочитать в пикселе (x, y).pixels[3] номер грани
171     gl.readPixels(x, y, 1, 1, gl.RGBA, gl.UNSIGNED_BYTE, pixels);
172
173     return pixels[3];
174 }
```

Начнем обсуждение с вершинного шейдера. В строке 6 объявляется переменная-атрибут `a_Face`, используемая для передачи номера грани, который затем будет «зашифрован» в альфа-компонент цвета грани в момент щелчка мышью. Номера граней определяются в `initVertexBuffers()` и просто отображаются на вершины граней. Это отображение описывается, начиная со строки 127. Например, вершины `v0-v1-v2-v3` определяют грань с номером 1, вершины `v0-v3-v4-v5` – грань с номером 2, и так далее. Так как в вершинный шейдер номер грани нужно передавать для каждой вершины, в строке 128, представляющей описание первой грани, указаны четыре единицы.

Если грань уже выбрана, `u_PickedFace` (объявляется в строке 8) сообщит вершинному шейдеру номер выбранной грани, что позволит ему изменить отображение грани.

В строке 12 вещественное значение переменной `a_Face` приводится к типу `int`. Так реализовано потому, что переменные-атрибуты не могут иметь тип `int` (см. главу 6, «Язык шейдеров OpenGL ES (GLSL ES)»). Если номер выбранной грани совпадает с номером текущей обрабатываемой грани, ей назначается белый цвет (строка 13). В противном случае используется собственный цвет грани. Если произошел щелчок мышью (то есть, переменная `u_PickedFace` имеет значение 0), значение `a_Face` вставляется в альфа-компонент значения цвета (строка 15).

Если теперь передать значение 0 в переменную `u_PickedFace` в момент щелчка мышью, грань будет нарисована со значением альфа-канала, совпадающим с номером грани. Переменная `u_PickedFace` инициализируется значением `-1` в строке 75. Так как у нас нет грани с номером `-1` (см. массив `faces` в строке 127), первоначально куб будет нарисован без выделения выбранной грани.

А теперь посмотрим, как работает обработчик событий. В строке 85 функции `checkFace()` передается значение переменной `u_PickedFace`. Функция `checkFace()` (начинается в строке 168) возвращает номер выбранной грани. В строке 168 в переменную `u_PickedFace` передается значение 0, сообщающее вершинному шейдеру, что выполнен щелчок мышью. Когда в следующей строке будет вызвана функция `draw()`, она перерисует объект и номер выбранной грани

окажется записанным в альфа-канал. В строке 171 проверяется значение пикселя в позиции шелчка, а в строке 173, обращением к `pixels[3]`, извлекается вставленный номер грани. (Это значение альфа-канала, которому соответствует индекс 3.) Этот номер возвращается вызывающему коду и затем используется в строках 86 и 87 для рисования куба. Остальную работу выполняет вершинный шейдер, как описывалось выше.

Эффект индикации на лобовом стекле (ИЛС)

Первоначально индикация на лобовом стекле была разработана в авиационной промышленности и предназначалась для формирования прозрачного изображения на остеклении кабины пилота, чтобы избавить его от необходимости переводить взгляд на панель приборов. Аналогичный эффект можно воссоздать в трехмерной графике для вывода текстовой информации в трехмерной сцене. В этом разделе мы рассмотрим программу, отображающую диаграмму и некоторую дополнительную информацию поверх трехмерного изображения (ИЛС), как показано на рис. 10.5.

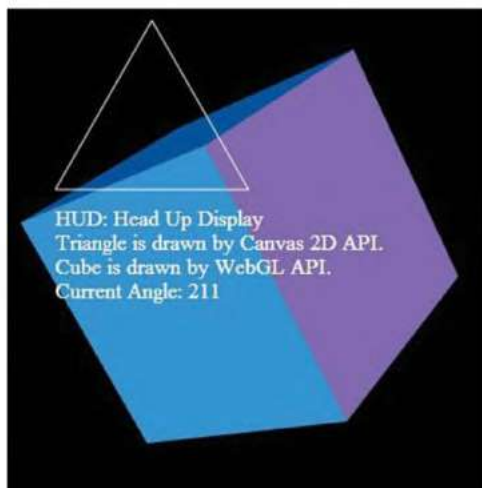


Рис. 10.5. ИЛС

Цель программы – нарисовать треугольник и вывести информацию о трехмерной сцене, включая текущий угол поворота куба (из `PickObject`), который будет меняться при вращении куба мышью.

Как реализовать ИЛС

Эффект ИЛС можно реализовать средствами HTML и с помощью функциональных возможностей элемента `<canvas>`, не прибегая к услугам системы WebGL. Делается это так:

1. В файле HTML готовятся два элемента `<canvas>`: один для вывода трехмерной графики с применением WebGL, и другой – для реализации ИЛС с применением стандартных функций. При этом элемент `<canvas>`, предназначенный для вывода ИЛС, помещается поверх элемента `<canvas>`, предназначенный для вывода трехмерной графики средствами WebGL.
2. В нижнем элементе `<canvas>` рисуется трехмерная графика с помощью WebGL API.
3. В верхнем элементе `<canvas>` выводится информация ИЛС с помощью стандартных функций `<canvas>`.

Как видите, все довольно просто, что еще раз доказывает мощь WebGL и возможность смешивания двух- и трехмерной графики в окне браузера. Рассмотрим, как этот подход реализован в программном коде.

Пример программы (HUD.html)

Так как для реализации ИЛС требуется добавить еще один элемент `<canvas>` в файл HTML, мы покажем содержимое файла `HUD.html` в листинге 10.4, где дополнения выделены жирным.

Листинг 10.4. HUD.html

```
1 <!DOCTYPE html>
2 <html lang="ja">
3   ...
4   <body onload="main()">
5     <canvas id="webgl" width="400" height="400" style="position:
6       absolute; z-index: 0" >
7       Please use a browser that supports "canvas"
8     </canvas>
9     <canvas id="hud" width="400" height="400" style="position:
10      absolute; z-index: 1"></canvas>
11   ...
12   <script src="HUD.js"></script>
13 </body>
14 </html>
```

Атрибут `style` определяет, как выглядит элемент и как он располагается на странице, что дает возможность разместить элемент `<canvas>` для ИЛС поверх элемента `<canvas>`, предназначенного для вывода трехмерной графики. Информация о стиле определяется как комбинация свойств и их значений, разделенных двоеточием, как, например, в строке 9: `style="position: absolute"`. Свойства в списке отделяются друг от друга точкой с запятой.

В этом примере используется свойство `position`, определяющее особенности расположения элементов, и свойство `z-index`, определяющее иерархические отношения между элементами.

Позицию элемента можно задать в абсолютных координатах, если указать значение `absolute` для свойства `position`. Если не указывать координаты, все эле-

менты со свойством стиля `position: absolute` будут помещены в одну и ту же позицию. Свойство `z-index` определяет порядок, отображения элементов, когда в одном и том же месте оказывается несколько элементов. Элемент с большим значением свойства `z-index` будет отображаться поверх элемента с меньшим значением этого свойства. В данном случае свойство `z-index` элемента `<canvas>` для ИЛС (строка 12) имеет значение 1.

В результате, оба элемента `<canvas>` будут помещены в одно и то же место на странице, и элемент `<canvas>` для ИЛС будет отображаться поверх элемента `<canvas>` с трехмерной графикой. По умолчанию элементы `<canvas>` имеют прозрачный фон, что очень удобно, так как мы можем видеть `<canvas>` с трехмерной графикой сквозь элемент `<canvas>` для ИЛС. Все, что будет нарисовано в элементе `<canvas>` для ИЛС, будет отображаться поверх трехмерных объектов, создавая эффект индикации на лобовом стекле.

Пример программы (HUD.js)

Теперь перейдем к файлу `HUD.js`, представленному в листинге 10.5. Здесь имеются два важных изменения, если сравнивать с программой `PickObject.js`:

1. Необходимо получить контекст отображения для элемента `<canvas>` ИЛС и использовать его для рисования.
2. Зарегистрировать обработчик щелчка мышью в элементе `<canvas>` для ИЛС, а не в элементе `<canvas>` для WebGL.

Первое изменение реализовано точно так же, как это делалось в главе 2, «Первые шаги в WebGL», когда мы рисовали двухмерные фигуры. Второе изменение обусловлено тем, что элемент `<canvas>` для трехмерной графики находится под элементом `<canvas>` для ИЛС, и потому не сможет получать события щелчка мышью. Вершинный и фрагментный шейдеры остались теми же, как в программе `PickObject.js`.

Листинг 10.5. HUD.js

```
1 // HUD.js
...
30 function main() {
31     // Получить ссылку на элемент <canvas>
32     var canvas = document.getElementById('webgl');
33     var hud = document.getElementById('hud');
34     ...
35     // Получить контекст отображения для WebGL
36     var gl = getWebGLContext(canvas);
37     // Получить контекст отображения 2-мерной графики
38     var ctx = hud.getContext('2d');
39     ...
40     // Зарегистрировать обработчик событий
41     hud.onmousedown = function(ev) { // Кнопка мыши нажата
42         ...
43         check(gl, n, x_in_canvas, y_in_canvas, currentAngle, u_Clicked,
```



```

    viewProjMatrix, u_MvpMatrix);

    ...
91 }
92
93 var tick = function() { // Начало рисования
94     currentAngle = animate(currentAngle);
95     draw2D(ctx, currentAngle); // Нарисовать 2-мерную графику
96     draw(gl, n, currentAngle, viewProjMatrix, u_MvpMatrix);
97     requestAnimationFrame(tick, canvas);
98 };
99 tick();
100 }

...
184 function draw2D(ctx, currentAngle) {
185     ctx.clearRect(0, 0, 400, 400); // Очистить <ИЛС>
186     // Нарисовать треугольник белыми линиями
187     ctx.beginPath(); // Начать рисование
188     ctx.moveTo(120, 10); ctx.lineTo(200, 150); ctx.lineTo(40, 150);
189     ctx.closePath();
190     ctx.strokeStyle = 'rgba(255, 255, 255, 1)'; // Определить цвет линий
191     ctx.stroke(); // Нарисовать треугольник
192     // Нарисовать белые буквы
193     ctx.font = '18px "Times New Roman"';
194     ctx.fillStyle = 'rgba(255, 255, 255, 1)'; // Определить цвет букв
195     ctx.fillText('HUD: Head Up Display', 40, 180);
196     ctx.fillText('Triangle is drawn by Hud API.', 40, 200);
197     ctx.fillText('Cube is drawn by WebGL API.', 40, 220);
198     ctx.fillText('Current Angle: ' + Math.floor(currentAngle), 40, 240);
199 }

```

Поскольку поток выполнения программы достаточно прямолинеен, начнем ее исследование со строки 30, где начинается определение функции `main()`. Прежде всего, в строке 33, программа получает ссылку на элемент `<canvas>` для ИЛС. Эта ссылка используется в строке 43 для получения контекста отображения двухмерной графики (см. главу 2). Обработчик события щелчка мышью регистрируется в элементе `<canvas>` для ИЛС (со значением атрибута `id="hud"`), а не в элементе `<canvas>` для WebGL, как это делалось в программе `PickObject.js`. Это обусловлено тем, что данные события будут поступать элементу `<canvas>` для ИЛС, который размещается поверх элемента `<canvas>` для WebGL.

Код, начиная со строки 93, реализует анимацию и вызывает функцию `draw2D()` в строке 95 для вывода информации ИЛС.

Определение самой функции `draw2D()` начинается в строке 184. В качестве параметров она принимает контекст рисования (`ctx`) и текущий угол поворота (`currentAngle`). Строка 185 очищает `<canvas>` для ИЛС вызовом метода `clearRect()`, который принимает координаты верхнего левого угла, ширину и высоту очищаемой области. В строках со 187 по 191 выполняется рисование треугольника. Треугольники рисуются иначе, чем прямоугольники, о которых рассказывалось в главе 2. Чтобы нарисовать треугольник требуется определить путь (контур) треугольника. В строках со 187 по 191 определяется путь, задается цвет

и затем осуществляется собственно рисование. В строках от 193 и ниже задается цвет текста и шрифт, а затем вызывается функция `fillText()`, которая в первом параметре принимает буквы, а во втором и третьем параметрах – координаты для вывода. Строка 198 выводит текущий угол поворота. В ней используется метод `Math.floor()`, округляющий вещественное число до меньшего целого значения. Строка 185 очищает элемент `<canvas>`, потому что каждый раз выводится новое значение угла поворота.

Отображение трехмерного объекта в веб-странице (3DoverWeb)

Отображение трехмерного объекта в веб-странице проще всего выполнить с помощью WebGL, по аналогии с примером ИЛС. Только в этом случае элемент `<canvas>` для WebGL должен располагаться поверх остальных элементов веб-страницы, а его фон должен быть прозрачным. На рис. 10.6 показан скриншот программы 3DoverWeb.

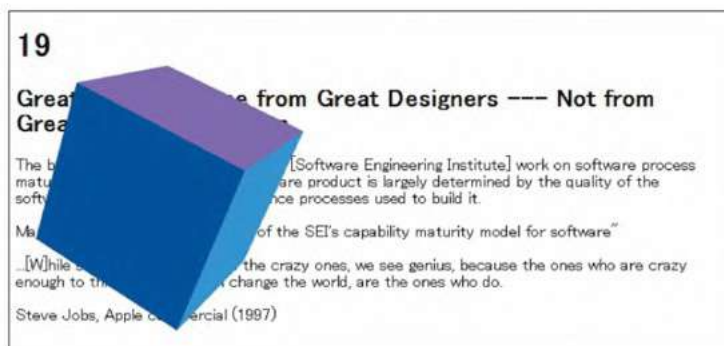


Рис. 10.6. 3DoverWeb¹

Программа `3DoverWeb.js` написана на основе `PickObject.js`. Единственное существенное отличие от прародительницы заключается в изменении цвета очистки – значение 1.0 для альфа-компонента изменилось на 0.0 в строке 55.

```
55 gl.clearColor(0.0, 0.0, 0.0, 0.0 );
```

Благодаря этому изменению, элемент `<canvas>` получил прозрачный фон и теперь сквозь него можно видеть содержимое веб-страницы. Вы можете поэкспериментировать со значением альфа-компонента; любое значение, отличное от 1.0, изменяет степень прозрачности и делает содержимое под элементом `<canvas>` видимым в большей или меньшей степени.

¹ Фоном для этой веб-страницы служит текст из книги «The Design of Design», Frederick P. Brooks Jr, Pearson (Фредерик П. Брукс, «Проектирование процесса проектирования: записки компьютерного эксперта», ISBN 978-5-8459-1792-8, Вильямс, 2013. – Прим. перев.).

Туман (атмосферный эффект)

Под термином «туман» (fog) в трехмерной графике подразумевается эффект размытия, или затуманивания объектов с увеличением расстояния до них. Этот эффект проявляется в любой среде, то есть, эффект размытия объектов под водой также обозначается этим термином. В этом разделе мы напишем программу Fog, воспроизводящую эффект затуманивания. Скриншот изображения, создаваемого программой, показан на рис. 10.7. Плотность тумана можно регулировать клавишами со стрелками вверх и вниз. Запустите эту программу и поэкспериментируйте с эффектом.

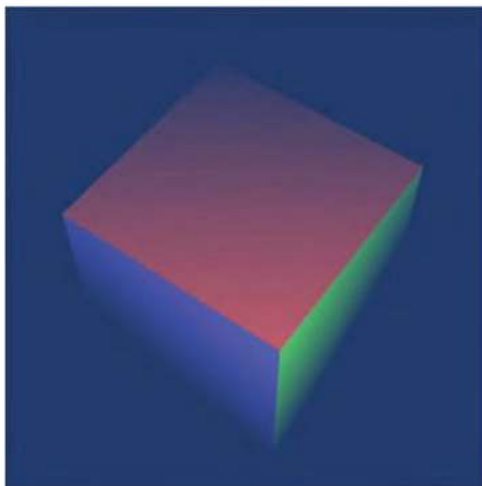


Рис. 10.7. Fog

Реализация эффекта тумана

Существуют разные способы вычисления степени затуманивания, но в этом разделе мы используем линейную зависимость (линейное нарастание плотности тумана), как наиболее простую в вычислениях. Линейный метод описывает нарастание плотности тумана от начальной точки (расстояние до точки, где объект начинает затуманиваться) до конечной (где объект становится полностью невидимым). Плотность тумана между этими двумя точками нарастает линейно. Имейте в виду, что под конечной точкой подразумевается не расстояние, где туман заканчивается, а расстояние, на котором туман становится настолько плотным, что скрывает все объекты. Коэффициент, описывающий, насколько ясно виден объект, мы будем называть **коэффициентом затуманивания**. В случае линейного нарастания плотности, он вычисляется в соответствии с формулой 10.1:

Формула 10.1.

$$\begin{aligned} \text{<коэффициент затуманивания>} = \\ \frac{(\text{<конечная точка>} - \text{<расстояние от точки наблюдения>})}{(\text{<конечная точка>} - \text{<начальная точка>})} \end{aligned}$$

где

$$(\text{начальная точка}) \leq (\text{расстояние от точки наблюдения}) \leq (\text{конечная точка})$$

Когда коэффициент затуманивания равен 1.0, объект виден ясно и отчетливо, а когда он равен 0.0, объект невидим полностью (см. рис. 10.8). Коэффициент затуманивания равен 1.0, когда $(\text{расстояние от точки наблюдения}) < (\text{начальная точка})$, и равен 0.0, когда $(\text{конечная точка}) < (\text{расстояние от точки наблюдения})$.

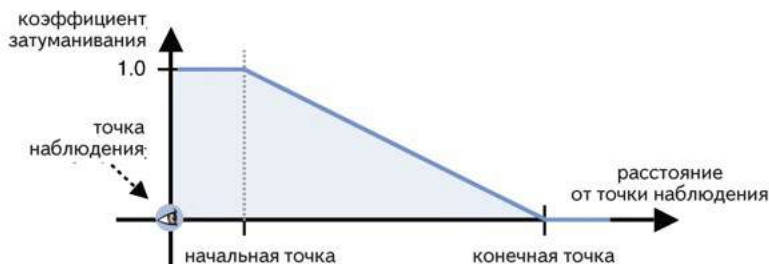


Рис. 10.8. Коэффициент затуманивания

На основании коэффициента затуманивания можно вычислять цвет фрагментов, в соответствии с формулой 10.2.

Формула 10.2.

$$\begin{aligned} \langle \text{цвет фрагмента} \rangle = & \\ & \langle \text{цвет грани} \rangle \times \langle \text{коэффициент затуманивания} \rangle + \\ & \langle \text{цвет тумана} \rangle \times (1 - \langle \text{коэффициент затуманивания} \rangle) \end{aligned}$$

Теперь посмотрим, как реализовать затуманивание в программном коде.

Пример программы (Fog.js)

В листинге 10.6 приводится исходный код программы Fog. Она (1) вычисляет в вершинном шейдере расстояние до объекта (вершины) от точки наблюдения и, исходя из этого расстояния, (2) во фрагментном шейдере вычисляет коэффициент затуманивания и цвет объекта. Обратите внимание, что в этой программе местоположение точки наблюдения задается в мировых координатах (см. приложение G, «Мировая и локальная системы координат»), соответственно коэффициент затуманивания также вычисляется с привязкой к системе мировых координат.

Листинг 10.6. Fog.js

```
1 // Fog.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4     ...
5     ...
6     ...
7     'uniform mat4 u_ModelMatrix;\n' +
8     'uniform vec4 u_Eye;\n' + // Точка наблюдения (мировые координаты)
9     'varying vec4 v_Color;\n' +
10    'varying float v_Dist;\n' +
```

```

11 'void main() {\n' +
12 '  gl_Position = u_MvpMatrix * a_Position;\n' +
13 '  v_Color = a_Color;\n' +
14 '    // Вычислить расстояние до каждой вершины от точки наблюдения      <-(1)
15 '  v_Dist = distance(u_ModelMatrix * a_Position, u_Eye);\n' +
16 '}\n';
17
18 // Фрагментный шейдер
19 var FSHADER_SOURCE =
20 ...
23 'uniform vec3 u_FogColor;\n' + // Цвет тумана
24 'uniform vec2 u_FogDist;\n' + // Начальная точка тумана, конечная точка
25 'varying vec4 v_Color;\n' +
26 'varying float v_Dist;\n' +
27 'void main() {\n' +
28 '    // Вычислить коэффициент затуманивания                               <-(2)
29 '    float fogFactor = clamp((u_FogDist.y - v_Dist) / (u_FogDist.y -
30 '                                     u_FogDist.x), 0.0, 1.0);\n' +
31 '    // u_FogColor * (1 - fogFactor) + v_Color * fogFactor
32 '    vec3 color = mix(u_FogColor, vec3(v_Color), fogFactor);\n' +
33 '    gl_FragColor = vec4(color, v_Color.a);\n' +
34 '}\n';
35 function main() {
36 ...
53 var n = initVertexBuffers(gl);
37 ...
59 // Цвет тумана
60 var fogColor = new Float32Array([0.137, 0.231, 0.423]);
61 // Расстояние [до начальной точки, до конечной точки]
62 var fogDist = new Float32Array([55, 80]);
63 // Местоположение точки наблюдения (в мировых координатах)
64 var eye = new Float32Array([25, 65, 35]);
65 ...
76 // Передать цвет тумана, расстояния и точку наблюдения
77 gl.uniform3fv(u_FogColor, fogColor); // Цвет тумана
78 gl.uniform2fv(u_FogDist, fogDist);   // Начальная и конечная точки
79 gl.uniform4fv(u_Eye, eye);           // Точка наблюдения
80
81 // Определить цвет очистки и включить удаление невидимых поверхностей
82 gl.clearColor( fogColor[0], fogColor[1], fogColor[2], 1.0);
83 ...
93 mvpMatrix.lookAt(eye[0], eye[1], eye[2], 0, 2, 0, 0, 1, 0);
94 ...
97 document.onkeydown = function(ev){keydown(ev, gl, n, u_FogDist, fogDist)};
98 ...

```

Вычисление расстояния от точки наблюдения до вершины выполняется в вершинном шейдере. Для этого координаты вершины преобразуются в мировые, с применением матрицы модели, и затем вызывается встроенная функция `distance()`, которой передаются координаты (мировые) точки наблюдения и координаты вершины. Функция `distance()` вычисляет расстояние между двумя точками в пространстве, заданными аргументами. Эти вычисления производятся

в строке 15, их результат сохраняется в переменной `v_Dist` и затем передается во фрагментный шейдер.

Фрагментный шейдер вычисляет цвет фрагмента с учетом затуманивания (строка 29), используя формулы 10.1 и 10.2. Необходимые для этого цвет тумана, начальная точка и конечная точка передаются в `uniform`-переменных `u_FogColor` и `u_FogDist` в строках 23 и 24. Компонент `u_FogDist.x` определяет начальную точку, а компонент `u_FogDist.y` – конечную.

Коэффициент затуманивания вычисляется в строке 29 по формуле 10.1. Функция `clamp()` – это встроенная функция; если значение первого ее параметра выходит за пределы диапазона, заданного вторым и третьим параметрами (в данном случае `[0.0 0.1]`), она возвращает значение, равное одному из концов диапазона. То есть, в данном случае, если значение первого параметра меньше 0.0, функция вернет 0.0, а если значение первого параметра больше 1.0, функция вернет 1.0. Если значение первого параметра попадает в диапазон, функция вернет его без изменений.

В строке 31, на основе коэффициента затуманивания, вычисляется цвет фрагмента. В этой реализации формулы 10.2 используется встроенная функция `mix()`, которая вычисляет выражение $x*(1 - z) + y*z$, где x – это первый параметр, y – второй и z – третий.

Функция `main()` в программном коде на JavaScript, определение которой начинается в строке 35, записывает исходные значения, необходимые для обсчета эффекта затуманивания, в соответствующие `uniform`-переменные.

Вы должны помнить, что кроме линейного существует и другие варианты расчетов эффекта затуманивания, например экспоненциальный, используемый в библиотеке OpenGL (см. книгу «OpenGL Programming Guide»). Вы с успехом можете использовать любой из этих вариантов, изменив лишь вычисления во фрагментном шейдере.

Использование значения `w` (`Fog_w.js`)

Вычисление расстояния в шейдере может отрицательно сказаться на производительности. Чтобы избежать этого, можно воспользоваться альтернативным методом, основанном на простой аппроксимации расстояния от точки наблюдения до объекта (вершины), задействовав значение `w` в векторе с координатами, преобразованными применением матрицы модели вида представления. В данном случае координаты подставляются в `gl_Position`. Четвертый компонент вектора `gl_Position`, который нигде явно не используется, – это значение `z` каждой вершины в системе видимых координат, умноженное на `-1`. Точка наблюдения в системе видимых координат всегда находится в ее начале, а взгляд всегда направлен вдоль оси `Z` в сторону отрицательных значений. Значение компонента `w`, можно использовать для аппроксимации расстояния.

Если реализовать вычисления в вершинном шейдере с применением компонента `w`, как показано в листинге 10.7, эффект затуманивания будет воспроизводиться точно так же, как в предыдущей программе.

Листинг 10.7. Fog_w.js

```
1 // Fog_w.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4
5     ...
6
7     'varying vec4 v_Color;\n' +
8     'varying float v_Dist;\n' +
9     'void main() {\n' +
10     '    gl_Position = u_MvpMatrix * a_Position;\n' +
11     '    v_Color = a_Color;\n' +
12     '    // Использовать отрицательное значение z вершины в системе видимых координат
13     '    v_Dist = gl_Position.w;\n' +
14     '}\n';
```

Создание круглой точки

В главе 2 мы написали программу, рисующую точку, чтобы помочь вам освоить основы шейдеров. Однако, чтобы сосредоточить все ваше внимание на работе шейдеров, мы рисовали точку не «круглой», а «квадратной», что намного проще. В этом разделе мы напишем аналогичную программу, `RoundedPoint`, которая рисует круглую точку (см. рис. 10.9).

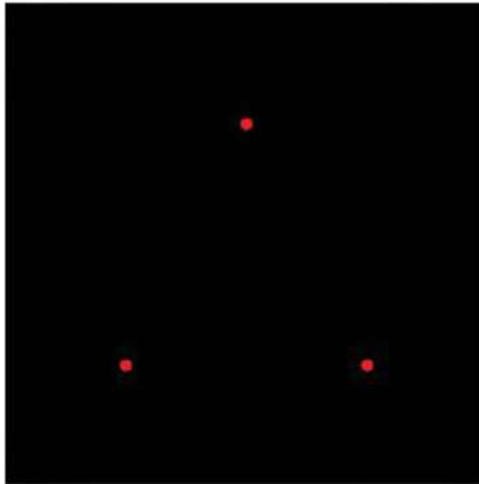


Рис. 10.9. `RoundedPoint`

Как нарисовать круглую точку

Чтобы нарисовать «круглую» точку, достаточно просто скруглить углы в «квадратной» точке. Эту операцию можно выполнить на этапе растеризации, который выполняется между вызовами вершинного и фрагментного шейдеров, как описывалось в главе 5, «Цвет и текстура». На этапе растеризации генерируется прямоугольник, состоящий из множества фрагментов, каждый из которых передается

фрагментному шейдеру. Если нарисовать эти фрагменты «как есть», получится прямоугольник. Поэтому нам нужно всего лишь изменить фрагментный шейдер, который будет рисовать только фрагменты, попадающие внутрь окружности, как показано на рис. 10.10.

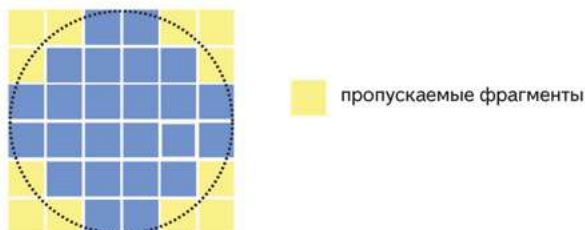


Рис. 10.10. Пропуск фрагментов с целью преобразовать прямоугольник в круг

Для этого требуется знать позиции всех фрагментов, созданных на этапе растеризации. В главе 5 мы исследовали программу, использующую встроенную переменную `gl_FragCoord` для передачи (ввода) данных фрагментному шейдеру. Однако, имеется еще одна похожая переменная – `gl_PointCoord` – которую можно задействовать для рисования круглой точки (см. табл. 10.1).

Таблица 10.1. Встроенные переменные для передачи данных фрагментному шейдеру

Тип и имя переменной	Описание
<code>vec4 gl_FragCoord</code>	Оконные координаты фрагмента
<code>vec2 gl_PointCoord</code>	Положение фрагмента в пределах нарисованной точки (от 0.0 до 1.0)

Переменная `gl_PointCoord` хранит координаты каждого фрагмента в диапазоне от (0.0, 0.0) до (1.0, 1.0), как показано на рис. 10.11. Чтобы закруглить углы, нужно просто пропустить фрагменты, выходящие за окружность с центром в точке (0.5, 0.5) и с радиусом 0.5. Выполнить пропуск фрагментов можно с помощью инструкции `discard`.

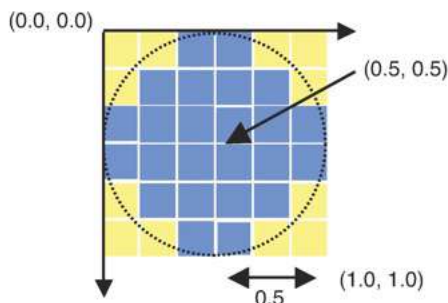


Рис. 10.11. Координаты в переменной `gl_PointCoord`

Пример программы (RoundedPoints.js)

В листинге 10.8 приводится исходный код обсуждаемой программы. Она написана на основе программы `MultiPoint.js`, созданной нами в главе 4, «Дополнительные преобразования и простая анимация», которая рисует несколько точек. Единственное отличие находится во фрагментном шейдере. Однако, для полноты картины, мы привели также вершинный шейдер.

Листинг 10.8. `RoundedPoint.js`

```
1 // RoundedPoints.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'void main() {\n' +
6   '   gl_Position = a_Position;\n' +
7   '   gl_PointSize = 10.0;\n' +
8   '}\n';
9
10 // Фрагментный шейдер
11 var FSHADER_SOURCE =
12   ...
13   'void main() {\n' + // Центр координат находится в точке (0.5, 0.5)
14   '   float dist = distance(gl_PointCoord, vec2(0.5, 0.5));\n' +
15   '   if(dist < 0.5) {\n' + // Радиус равен 0.5
16   '       gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);\n' +
17   '   } else { discard; }\n' +
18   '}\n';
19
20 ...
21
22 function main() {
23   ...
24   gl.drawArrays(gl.POINTS, 0, n);
25 }
```

Главным отличием этой версии программы являются вычисления, начиная со строки 16, где определяется, какие фрагменты должны быть пропущены. Переменная `gl_PointCoord` хранит координаты фрагмента (в диапазоне от 0.0 до 1.0), с центром координат в точке (0.5, 0.5). Соответственно, чтобы закруглить квадратную точку, нужно выполнить следующее:

1. Вычислить расстояние от центра (0.5, 0.5) до каждого фрагмента.
2. Нарисовать только те фрагменты, расстояние до которых меньше 0.5.

Расстояние от центра в программе `RoundedPoint.js` вычисляется в строке 16. Здесь просто вычисляется расстояние от точки с координатами (0.5, 0.5) до точки с координатами, хранящимися в `gl_PointCoord`. Так как переменная `gl_PointCoord` имеет тип `vec2`, координаты (0.5, 0.5) нужно также привести к типу `vec2` перед передачей их функции `distance()`.

После того, как расстояние будет вычислено, оно используется в строке 17, где сравнивается со значением 0.5 (иными словами, определяется – находится ли фрагмент внутри окружности). Если фрагмент находится внутри окружности,

шейдер рисует его в строке 18, используя цвет `gl_FragColor`. В противном случае, в строке 19, выполняется инструкция `discard`, которая заставляет WebGL пропустить этот фрагмент.

Альфа-смешивание

Значение альфа-канала управляет прозрачностью рисуемых объектов. Если установить значение альфа-канала равным 0.5, объект станет полупрозрачным и через него станут частично видимы другие объекты. По мере приближения значения альфа-канала к 0.0, находящиеся позади объекты будут видимы все отчетливее. Если вы попытаетесь проделать это сами, вы увидите, что в действительности с уменьшением значения альфа-канала объекты будут становиться все более и более белыми. Это объясняется тем, что по умолчанию WebGL использует одно и то же альфа-значение и для объектов, и для элемента `<canvas>`. В примерах программ, которые приводились до сих пор, веб-страница «позади» элемента `<canvas>` имеет белый цвет, и именно она начинает «просвечивать» сквозь полупрозрачные объекты и область рисования.

Давайте теперь напомним программу, демонстрирующую использование альфа-смешивания для получения желаемого результата. Функция, позволяющая использовать значение альфа-канала, называется **функцией альфа-смешивания** (alpha blending), или просто **функцией смешивания** (blending). Данная функция уже встроена в систему WebGL, поэтому ее нужно просто активировать, сообщив системе WebGL, что она должна начать использовать указываемые программой значения альфа-канала.

Как реализовать альфа-смешивание

Чтобы активировать функцию альфа-смешивания и включить ее в работу, необходимо выполнить следующие два шага.

1. Активировать функцию альфа-смешивания:

```
gl.enable(gl.BLEND);
```

2. Указать, как должна действовать функция смешивания:

```
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
```

О параметрах, определяющих, как должна действовать функция смешивания мы поговорим позже, а пока поэкспериментируем с программой. Здесь мы снова воспользуемся программой `LookAtTrianglesWithKey_ViewVolume`, описанной в главе 7, «Вперед, в трехмерный мир». Как показано на рис. 10.12, эта программа рисует три треугольника и позволяет менять точку наблюдения с помощью клавиш со стрелками.

Добавим в код реализацию шагов 1 и 2, укажем значение 0.4 альфа-канала в значениях цвета треугольников и дадим получившейся программе имя `LookAtBlendedTriangles`. На рис. 10.13 показан эффект, который воспроизводит эта программа. Как видите, все три треугольника стали полупрозрачными и сквозь

них можно видеть, что находится за ними. Смещая точку наблюдения клавишами со стрелками, можно видеть, как постоянно происходит смешивание изображений треугольников.



Рис. 10.12.

LookAtTrianglesWithKeys_ViewVolume



Рис. 10.13.

LookAtBlendedTriangles

А теперь исследуем саму программу.

Пример программы (*LookAtBlendedTriangles.js*)

Исходный код программы `LookAtBlendedTriangles.js` представлен в листинге 10.9. В нем, по сравнению с листингом программы, взятой за основу, изменились строки с 51 по 54 и в определении массива с информацией о цвете (в функции `initVertexBuffer()`, строки с 81 по 91) было добавлено значение альфа-канала (0.4). Соответственно изменились значения параметров `size` и `stride` в вызове `gl.vertexAttribPointer()`.

Листинг 10.9. `LookAtBlendedTriangles.js`

```
1 // LookAtBlendedTriangles.js
2 // Оригинал: LookAtTrianglesWithKey_ViewVolume.js
...
25 function main() {
    ...
43   var n = initVertexBuffers(gl);
    ...
51   // Активировать альфа-смешивание
52   gl.enable (gl.BLEND);
53   // Указать, как должна действовать функция смешивания
54   gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
    ...
75   draw(gl, n, u_ViewMatrix, viewMatrix);
```

```

76 }
77
78 function initVertexBuffers(gl) {
79     var verticesColors = new Float32Array([
80         // Координаты вершин и цвет(RGBA)
81         0.0, 0.5, -0.4, 0.4, 1.0, 0.4, 0.4,
82         -0.5, -0.5, -0.4, 0.4, 1.0, 0.4, 0.4,
83         ...
91         0.5, -0.5, 0.0, 1.0, 0.4, 0.4, 0.4,
92     ]);
93     var n = 9;
94     ...
127     return n;
128 }

```

Как должна действовать функция смешивания

Давайте исследуем функцию `gl.blendFunc()`, чтобы понять, как ее использовать для достижения эффекта смешивания. В смешивании участвуют два цвета: цвет, который будет подмешиваться (исходный цвет) и цвет, в который будет осуществляться подмешивание (целевой цвет). Например, когда рисуется один треугольник поверх другого, цвет уже нарисованного треугольника является целевым цветом, а цвет треугольника, рисуемого сверху, является исходным цветом.

`gl.blendFunc(src_factor, dst_factor)`

Определяет, как должно выполняться смешивание исходного цвета с целевым. Результирующий цвет вычисляется по следующей формуле:

$$\text{цвет (RGB)} = \langle \text{исходный цвет} \rangle \times \text{src_factor} + \langle \text{целевой цвет} \rangle \times \text{dst_factor}$$

Параметры	<code>src_factor</code>	Определяет коэффициент для исходного цвета (см. табл. 10.2).
	<code>dst_factor</code>	Определяет коэффициент для целевого цвета (см. табл. 10.2).
Возвращаемое значение	нет	
Ошибки	<code>INVALID_ENUM</code>	Параметры <code>src_factor</code> и <code>dst_factor</code> имеют недопустимые значения.

Таблица 10.2. Константы, которые могут использоваться в качестве значений параметров `src_factor` и `dst_factor`²

Константа	Множитель для R	Множитель для G	Множитель для B
<code>gl.SRC_COLOR</code>	R_s	G_s	B_s
<code>gl.ONE_MINUS_SRC_COLOR</code>	$(1 - R_s)$	$(1 - G_s)$	$(1 - B_s)$
<code>gl.DST_COLOR</code>	R_d	G_d	B_d

² Константы `gl.CONSTANT_COLOR`, `gl.ONE_MINUS_CONSTANT_COLOR`, `gl.CONSTANT_ALPHA` и `gl.ONE_MINUS_CONSTANT_ALPHA` были удалены из OpenGL.

Константа	Множитель для R	Множитель для G	Множитель для B
<code>gl.ONE_MINUS_DST_COLOR</code>	$(1 - R_d)$	$(1 - G_d)$	$(1 - B_d)$
<code>gl.SRC_ALPHA</code>	A_s	A_s	A_s
<code>gl.ONE_MINUS_SRC_ALPHA</code>	$(1 - A_s)$	$(1 - A_s)$	$(1 - A_s)$
<code>gl.DST_ALPHA</code>	A_d	A_d	A_d
<code>gl.ONE_MINUS_DST_ALPHA</code>	$(1 - A_d)$	$(1 - A_d)$	$(1 - A_d)$
<code>gl.SRC_ALPHA_SATURATE</code>	$\min(A_s, A_d)$	$\min(A_s, A_d)$	$\min(A_s, A_d)$

(R_s , G_s , B_s , A_s) – это исходный цвет, а (R_d , G_d , B_d , A_d) – целевой цвет.

В данной программе используются следующие значения:

```
54 gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
```

Например, если исходный цвет – полупрозрачный зеленый (0.0, 1.0, 0.0, 0.4) и целевой цвет – желтый (1.0, 1.0, 0.0, 1.0), параметр `src_factor` получит значение альфа-канала 0.4, а параметр `dst_factor` получит значение $(1 - 0.4) = 0.6$. Вычисления показаны на рис. 10.14.

`color(RGB) = исходный цвет * src_factor + целевой цвет * dst_factor`

	R	G	B		R	G	B
source color	0.0	1.0	0.0	destination color	1.0	1.0	0.0
* src_factor	0.4	0.4	0.4	* dst_factor	0.6	0.6	0.6
	0.0	0.4	0.0		0.6	0.6	0.0
				Смешанный цвет 0.6 1.0 0.0			

Рис. 10.14. Вычисления для `gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA)`

Поэкспериментируйте с другими допустимыми значениями параметров `src_factor` и `dst_factor`. Однако на практике часто используется кумулятивное смешивание. Результат такого смешивания получается более ярким, чем оригинал. Этот прием можно использовать, например, для достижения эффекта подсветки от взрыва.

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE);
```

Альфа-смешивание для трехмерных объектов (*BlendedCube.js*)

А теперь исследуем эффект альфа-смешивания на примере представителя трехмерных объектов, куба, сделав его полупрозрачным. Здесь мы повторно воспользуемся программой `ColoredCube` из главы 7, и на ее основе создадим программу `BlendedCube`, в которую добавим два шага, необходимых для организации смешивания (см. листинг 10.10).

Листинг 10.10. BlendedCube.js

```
1 // BlendedCube.js
...
47 // Определить цвет для очистки и включить удаление скрытых поверхностей
48 gl.clearColor(0.0, 0.0, 0.0, 1.0);
49 gl.enable(gl.DEPTH_TEST);
50 // Активировать функцию альфа-смешивания
51 gl.enable(gl.BLEND);
52 // Определить, как должно производиться смешивание
53 gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
```

К сожалению, если запустить программу в текущем ее виде, мы не увидим ожидаемый результат (справа на рис. 10.15); вместо этого мы увидим изображение, показанное слева, ничем не отличающееся от изображения, которое создает программа ColoredCube из главы 7.

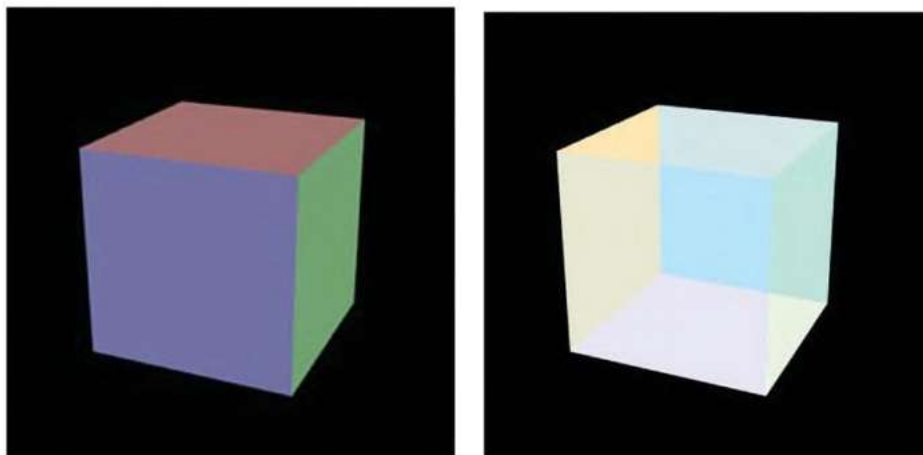


Рис. 10.15. BlendedCube

Это объясняется тем, что мы активировали механизм удаления скрытых поверхностей в строке 49. Смешивание имеет место, только когда осуществляется рисование поверхностей. С активированным механизмом удаления скрытых поверхностей рисование таких поверхностей не производится, поэтому не происходит и смешивание их цветов. По этой причине мы не наблюдаем ожидаемый эффект. Чтобы исправить эту проблему, можно просто закомментировать строку 49.

```
48 gl.clearColor(0.0, 0.0, 0.0, 1.0);
49 //gl.enable(gl.DEPTH_TEST);
50 // Активировать функцию альфа-смешивания
51 gl.enable(gl.BLEND);
```

Рисование при наличии прозрачных и непрозрачных объектов

Это самое простое решение, но его нельзя признать удовлетворительным, потому что, как мы видели в главе 7, удаление скрытых поверхностей часто просто необходимо для правильного отображения трехмерных сцен.

Решить эту проблему можно переключением механизма удаления скрытых поверхностей во время рисования объектов.

1. Включить удаление скрытых поверхностей.

```
gl.enable(gl.DEPTH_TEST);
```

2. Нарисовать непрозрачные объекты (имеющие значение 1.0 альфа-канала).
3. Сделать буфер глубины (см. главу 7), используемый для удаления скрытых поверхностей, доступным только для чтения.

```
gl.depthMask(false);
```

4. Нарисовать все прозрачные объекты (имеющие значение альфа-канала меньше 1.0). Обратите внимание, что они должны быть отсортированы по глубине и рисоваться в порядке от наиболее удаленных к наиболее близким.

5. Сделать буфер глубины доступным для чтения и записи.

```
gl.depthMask(true);
```

Если полностью запретить удаление скрытых поверхностей, прозрачные объекты, находящиеся за непрозрачными не будут скрыты. Именно поэтому необходимо осуществлять управление буфером глубины с помощью функции `gl.depthMask()`, которая описывается ниже.

`gl.depthMask(mask)`

Разрешает или запрещает запись в буфер глубины

Параметры	<code>mask</code>	Определяет доступность буфера глубины для записи. Если в параметре <code>mask</code> передать значение <code>false</code> , функция сделает буфер глубины недоступным для записи.
------------------	-------------------	---

Возвращаемое значение	нет
------------------------------	-----

Ошибки	нет
---------------	-----

В общих чертах с буфером глубины мы познакомились в главе 7. В буфер глубины записываются z-значения фрагментов (нормализованные в диапазоне от 0.0 до 1.0). Например, пусть есть два треугольника, расположенные друг за другом, и первым рисуется треугольник, находящийся ближе к точке наблюдения. Сначала в буфер глубины запишется z-значение ближнего треугольника. Затем, когда начнется рисование дальнего треугольника, механизм удаления скрытых поверхностей сравнит z-значение рисуемого фрагмента с z-значением фрагмента, уже имеющимся в буфере глубины. И только если z-значение фрагмента, рисуемого в данный момент, окажется меньше z-значения в буфере глубины (то есть, ближе

к точке наблюдения), рисуемый фрагмент попадет в буфер цвета. Этот алгоритм гарантирует удаление скрытых поверхностей. Таким образом, по окончании рисования, в буфере глубины останется z-значение фрагмента поверхности, наиболее близкой к точке наблюдения.

Непрозрачные объекты будут нарисованы в буфере цвета в правильном порядке, благодаря действию механизма удаления скрытых поверхностей, в ходе выполнения шагов 1 и 2, а z-значение, определяющее порядок, будет записано в буфер глубины. Рисование прозрачных объектов в буфере цвета с использованием их z-значений осуществляется в ходе выполнения шагов 3, 4 и 5, поэтому скрытые поверхности прозрачных объектов, оказавшиеся позади непрозрачных объектов, будут удалены. Это приводит к правильному отображению прозрачных и непрозрачных объектов.

Переключение шейдеров

Во всех примерах программ, приводившихся в этой книге, для рисования использовался единственный вершинный шейдер и единственный фрагментный шейдер. Если все объекты, образующие сцену можно нарисовать единственным шейдером, то никаких проблем не возникает. Однако, если каждый объект должен рисоваться по-своему, вам может потребоваться усложнить шейдеры, чтобы добиться желаемого результата. Избавиться от лишних сложностей можно, если создать несколько шейдеров и затем переключать их по мере необходимости. В этом разделе мы напишем программу `ProgramObject`, которая рисует два куба: один окрашен в единый цвет, а на другой наложена текстура. Изображение, формируемое программой, показано на рис. 10.16.

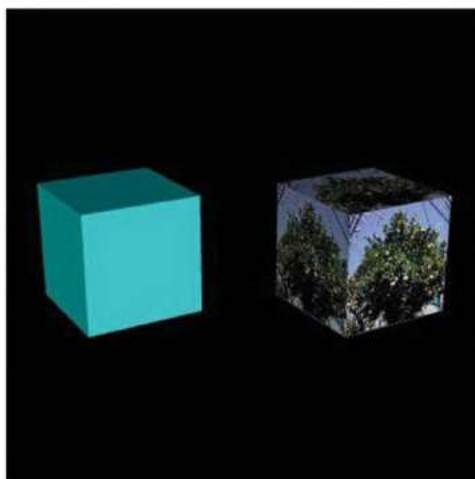


Рис. 10.16. `ProgramObject`

Эта программа служит также примером реализации затенения текстурированных объектов.

Как реализовать переключение шейдеров

Реализовать переключение шейдеров можно, создав требуемое число объектов программ, как описано в главе 9, «Иерархические объекты», и переключая их перед рисованием. Собственно переключение выполняется вызовом функции `gl.useProgram()`. Так как такой подход требует ручного управления объектами шейдеров, при его использовании уже нельзя будет воспользоваться удобной функцией `initShaders()`. Однако можно использовать функцию `createProgram()` из библиотеки `cuon-utils.js`, которая вызывается из `initShaders()`.

Ниже описывается порядок действия примера программы, которую мы исследуем в этом разделе. Каждое действие она выполняет дважды, поэтому список выглядит длинным, но сам код достаточно прост:

1. Подготовить шейдеры для рисования объекта, закрашиваемого одним цветом.
2. Подготовить шейдеры для рисования объекта с наложенной текстурой.
3. Создать объект программы с шейдерами, созданными на шаге 1, вызовом `createProgram()`.
4. Создать объект программы с шейдерами, созданными на шаге 2, вызовом `createProgram()`.
5. Задействовать объект программы, созданный на шаге 3, вызовом `gl.useProgram()`.
6. Разрешить буферный объект после сохранения в нем переменных-атрибутов.
7. Нарисовать куб, закрашиваемый одним цветом.
8. Задействовать объект программы, созданный на шаге 4, вызовом `gl.useProgram()`.
9. Разрешить буферный объект после сохранения в нем переменных-атрибутов.
10. Нарисовать куб с наложенной текстурой.

А теперь рассмотрим саму программу.

Пример программы (ProgramObject.js)

Основной код программы, реализующий шаги с 1 по 4, представлен в листинге 10.11. В этом фрагменте демонстрируется создание двух вершинных и двух фрагментных шейдеров: `SOLID_VSHADER_SOURCE` (строка 3) и `SOLID_FSHADER_SOURCE` (строка 19) для рисования однотонного объекта, и `TEXTURE_VSHADER_SOURCE` (строка 29) и `TEXTURE_FSHADER_SOURCE` (строка 46) для рисования объекта с текстурой. Так как основной интерес в этом примере для нас представляет реализация переключения объектов программ, мы опустили код шейдеров.

Листинг 10.11. ProgramObject (реализация шагов с 1 по 4)

```
1 // ProgramObject.js
2 // Вершинный шейдер для рисования объекта одним цветом <- {1}
```



```

3 var SOLID_VSHADER_SOURCE =
...
18 // Фрагментный шейдер для рисования объекта одним цветом
19 var SOLID_FSHADER_SOURCE =
...
28 // Вершинный шейдер для рисования объекта с текстурой <- (2)
29 var TEXTURE_VSHADER_SOURCE =
...
45 // Фрагментный шейдер для рисования объекта с текстурой
46 var TEXTURE_FSHADER_SOURCE =
...
58 function main() {
...
69 // Инициализировать шейдеры
70 var solidProgram = createProgram (gl, SOLID_VSHADER_SOURCE,
    ↗SOLID_FSHADER_SOURCE); <- (3)
71 var texProgram = createProgram (gl, TEXTURE_VSHADER_SOURCE,
    ↗TEXTURE_FSHADER_SOURCE); <- (4)
...
77 // Получить ссылки на переменные в объектах программ для одноцветного куба
78 solidProgram.a_Position = gl.getAttribLocation(solidProgram, 'a_Position');
79 solidProgram.a_Normal = gl.getAttribLocation(solidProgram, 'a_Normal');
...
83 // Получить ссылки на переменные в объектах программ для куба с текстурой
84 texProgram.a_Position = gl.getAttribLocation(texProgram, 'a_Position');
85 texProgram.a_Normal = gl.getAttribLocation(texProgram, 'a_Normal');
...
89 texProgram.u_Sampler = gl.getUniformLocation(texProgram, 'u_Sampler');
...
99 // Определить информацию о вершинах
100 var cube = initVertexBuffers(gl, solidProgram);
...
106 // Определить текстуру
107 var texture = initTextures(gl, texProgram);
...
122 // Начать рисование
123 var currentAngle = 0.0; // Текущий угол поворота (в градусах)
124 var tick = function() {
125     currentAngle = animate(currentAngle); // Изменить угол поворота
    ...
128     // Нарисовать одноцветный куб
129     drawSolidCube(gl, solidProgram, cube, -2.0, currentAngle, viewProjMatrix);
130     // Нарисовать куб с текстурой
131     drawTexCube(gl, texProgram, cube, texture, 2.0, currentAngle,
        ↗viewProjMatrix);
132
133     window.requestAnimationFrame(tick, canvas);
134 };
135 tick();
136 }
137
138 function initVertexBuffers(gl, program) {
    ...
148     var vertices = new Float32Array([ // Координаты вершин

```

```

149     1.0, 1.0, 1.0, -1.0, 1.0, 1.0, -1.0,-1.0, 1.0, 1.0,-1.0, 1.0,
150     1.0, 1.0, 1.0, 1.0,-1.0, 1.0, 1.0,-1.0,-1.0, 1.0, 1.0,-1.0,
151     ...
154     1.0,-1.0,-1.0, -1.0,-1.0,-1.0, -1.0, 1.0,-1.0, 1.0, 1.0,-1.0
155   });
156
157   var normals = new Float32Array([ // Нормали
158     ...
164   ]);
165
166   var texCoords = new Float32Array([ // Координаты текстуры
167     ...
173   ]);
174
175   var indices = new Uint8Array([ // Индексы для вершин
176     ...
182   ]);
183
184   var o = new Object(); // Использовать Object, чтобы вернуть буферные объекты
185
186   // Записать информацию о вершинах в буферный объект
187   o.vertexBuffer = initArrayBufferForLaterUse(gl, vertices, 3, gl.FLOAT);
188   o.normalBuffer = initArrayBufferForLaterUse(gl, normals, 3, gl.FLOAT);
189   o.texCoordBuffer = initArrayBufferForLaterUse(gl, texCoords, 2, gl.FLOAT);
190   o.indexBuffer = initElementArrayBufferForLaterUse(gl, indices,
191                                                     gl.UNSIGNED_BYTE);
192
193   ...
194   o.numIndices = indices.length;
195   ...
199   return o;
200 }

```

Начнем с JavaScript-функции `main()`. Сначала программа создает по программному объекту для каждой пары шейдеров вызовом функции `createProgram()` (в строках 70 и 71), которая принимает те же аргументы, что и функция `initShaders()`, и возвращает объект программы. Ссылки на эти объекты сохраняются в переменных `solidProgram` и `texProgram`. Затем программа получает ссылки на переменные-атрибуты и `uniform`-переменные в каждом из шейдеров (строки с 78 по 89). Ссылки сохраняются в соответствующих свойствах объектов программ, как мы делали это в примере `MultiJointModel_segment.js`. И снова мы воспользовались особенностью JavaScript, позволяющей свободно добавлять в объекты свойства любых типов.

Далее, вызовом `initVertexBuffer()` в строке 100, информация о вершинах сохраняется в буферном объекте. Чтобы нарисовать одноцветный объект, шейдеру необходимы: (1) координаты вершин, (2) нормали и (3) индексы. Кроме того, шейдеру, рисующему объект с текстурой, необходимы координаты текстуры. Функция `initVertexBuffer()` инициализирует все эти данные в буферных объектах, которые перед рисованием будут связаны с соответствующими переменными-атрибутами.

Инициализация координат вершин выполняется функцией `initVertexBuffer()`, начиная со строки 148, нормалей – начиная со строки 157,

координат текстуры – со строки 166, и индексов – со строки 175. В строке 184 создается объект (о) типа `Object`. Затем в свойствах этого объекта сохраняются буферные объекты (строки со 187 по 190). Буферные объекты можно было бы создать как глобальные переменные, но это привело бы к захламлению глобального пространства имен и ухудшило бы удобочитаемость исходного кода программы. Задействовав свойства объекта, можно управлять всеми четырьмя буферными объектами, используя единственный объект о.³

Для создания каждого буферного объекта мы использовали функцию `initArrayBufferForLaterUse()`, которая описана в обсуждении примера `MultiJointModel_segment.js`. Эта функция записывает информацию о вершинах в буферный объект, но не присваивает его переменной-атрибуту. Для облегчения понимания, мы дали свойствам объекта описательные имена. Строка 199 возвращает объект о вызывающей программе.

После возврата в функцию `main()`, вызовом `initTextures()` в строке 107 выполняется настройка текстуры, и на этом этапе все исходные данные готовы к рисованию двух кубов. Сначала мы рисуем одноцветный куб, с помощью `drawSolidCube()` (строка 129), а затем куб с текстурой, с помощью `drawTexCube()` (строка 131). В листинге 10.12 приводится реализация остальных шагов, выполняемых программой (с 5 по 10).

Листинг 10.12. `ProgramObject.js` (реализация шагов с 5 по 10)

```
236 function drawSolidCube(gl, program, o, x, angle, viewProjMatrix) {
237   gl.useProgram(program); // Задействовать этот объект программы    <-(5)
238
239   // Присвоить буферные объекты и разрешить присваивание            <-(6)
240   initAttributeVariable(gl, program.a_Position, o.vertexBuffer);
241   initAttributeVariable(gl, program.a_Normal, o.normalBuffer);
242   gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, o.indexBuffer);
243
244   drawCube(gl, program, o, x, angle, viewProjMatrix); // Нарисовать  <-(7)
245 }
246
247 function drawTexCube(gl, program, o, texture, x, angle, viewProjMatrix) {
248   gl.useProgram(program); // Задействовать этот объект программы    <-(8)
249
250   // Присвоить буферные объекты и разрешить присваивание            <-(9)
251   initAttributeVariable(gl, program.a_Position, o.vertexBuffer);
252   initAttributeVariable(gl, program.a_Normal, o.normalBuffer);
253   initAttributeVariable(gl, program.a_TexCoord, o.texCoordBuffer);
254   gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, o.indexBuffer);
255
256   // Связать объект текстуры с текстурным слотом 0
257   gl.activeTexture(gl.TEXTURE0);
258   gl.bindTexture(gl.TEXTURE_2D, texture);
259
260   drawCube(gl, program, o, x, angle, viewProjMatrix); // Нарисовать <-(10)
```

³ Мы использовали объект (о), чтобы упростить описание. Однако в подобных ситуациях предпочтительнее определять свой отдельный тип, предназначенный для работы с буферными объектами.


```
261 }
262
263 // Присваивает буферные объекты и разрешает присваивание
264 function initAttributeVariable(gl, a_attribute, buffer) {
265     gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
266     gl.vertexAttribPointer(a_attribute, buffer.num, buffer.type, false, 0, 0);
267     gl.enableVertexAttribArray(a_attribute);
268 }
...
275 function drawCube(gl, program, o, x, angle, viewProjMatrix) {
276     // Вычислить матрицу модели
...
281     // Вычислить матрицу преобразований для нормалей
...
286     // Вычислить матрицу модели вида проекции
...
291     gl.drawElements(gl.TRIANGLES, o.numIndices, o.indexBuffer.type, 0);
292 }
```

Определение `drawSolidCube()` начинается в строке 236. Она вызывает `gl.useProgram()` (строка 237), чтобы сообщить системе WebGL, что та должна использовать объект программы (`solidProgram`), указанный в аргументе. После этого можно приступить к рисованию с использованием `solidProgram`. В `initAttributeVariable()` буферные объекты с координатами вершин и нормальными присваиваются переменным-атрибутам (строки 240 и 241). Определение этой функции начинается в строке 264. Строка 242 связывает буферный объект, хранящий индексы, с типом `gl.ELEMENT_ARRAY_BUFFER`. После выполнения подготовительных операций, в строке 244 вызывается функция `drawCube()`, которая выполняет рисование с помощью `gl.drawElements()`, вызывая ее в строке 291.

В строке 247 начинается определение функции `drawTexCube()`, которая выполняет те же самые шаги, что и `drawSolidCube()`. Дополнительно в строке 253 она выполняет присваивание буферного объекта с координатами текстуры переменной-атрибуту, и в строках 257 и 258 связывает объект текстуры с текстурным слотом 0. Фактическое рисование выполняется вызовом `drawCube()`, как и в функции `drawSolidCube()`.

Овладев этим простым приемом, вы сможете переключаться между любым числом объектов программ и использовать самые разные эффекты в одной сцене.

Использование нарисованного изображения в качестве текстуры

В компьютерной графике существует один интересный прием: сначала нарисовать некоторую трехмерную сцену, а затем использовать получившееся изображение в качестве текстуры. Фактически, обладая такой возможностью, мы можем генерировать изображения, что называется «на лету». Это означает, что нет необходимости загружать изображения из сети и мы можем применять любые эффекты (такие

как размытие и уменьшение глубины резкости) перед отображением изображения. Этот прием можно также использовать для затенения, о чем мы поговорим в следующем разделе. Здесь мы напишем программу `FramebufferObject`, накладывающую на прямоугольник текстуру с изображением вращающегося куба. Результат работы программы представлен на рис. 10.17.

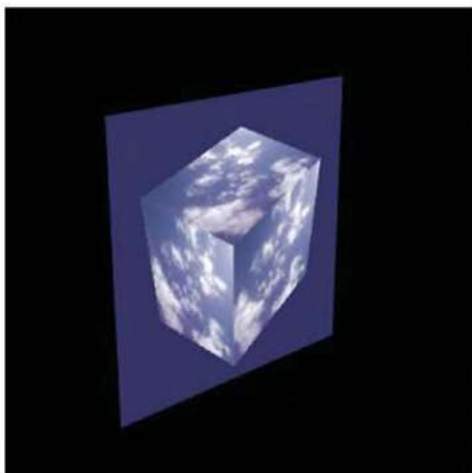


Рис. 10.17. `FramebufferObject`

Если запустить эту программу, можно увидеть прямоугольник с текстурой, изображающей вращающийся куб, текстурированный изображением летнего неба. Самое важное здесь, что вращающийся куб в прямоугольнике – это не видеоролик, подготовленный заранее, а анимация, созданная средствами WebGL в реальном времени. Эффект получается впечатляющий! Так давайте посмотрим, что нужно сделать, чтобы добиться его.

Объект буфера кадра и объект буфера отображения

По умолчанию система WebGL рисует изображения с использованием буфера цвета и, когда используется механизм удаления скрытых поверхностей, буфер глубины. Окончательное изображение формируется в буфере цвета.

Объект буфера кадра (`framebuffer object`) – это альтернативный механизм, который можно использовать вместо буфера цвета и/или буфера глубины (см. рис. 10.18). В отличие от буфера цвета, изображение, формируемое в буфере кадра, не отображается непосредственно в элементе `<canvas>`. Благодаря этому буфер кадра можно использовать для дополнительной обработки перед выводом изображения на экран. В нем также можно формировать изображение, которое будет играть роль текстуры. Этот прием часто называют **рисованием в памяти** (`offscreen drawing`).

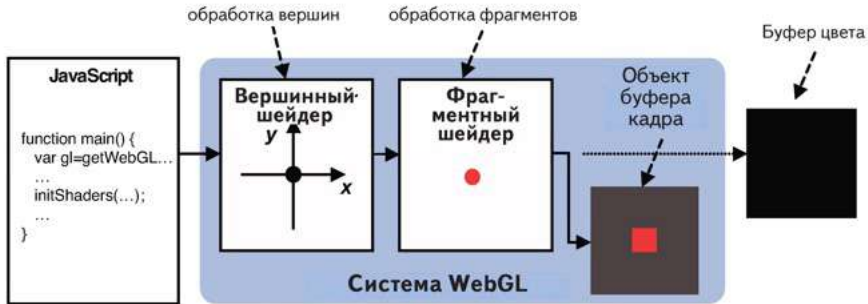


Рис. 10.18. Объект буфера кадра

На рис. 10.19 показано внутреннее устройство объекта буфера кадра, который может служить заменой буфера цвета и буфера глубины. Как видите, рисование выполняется не в самом буфере кадра, а в объектах, на которые он ссылается. Эти объекты подключаются к буферу кадра в виде ссылок. **Ссылка на буфер цвета** определяет область, где будет осуществляться рисование, которая является заменой буфера цвета. **Ссылка на буфер глубины** и **ссылка на буфер трафарета** определяют области, являющиеся заменой буфера глубины и буфера трафарета.

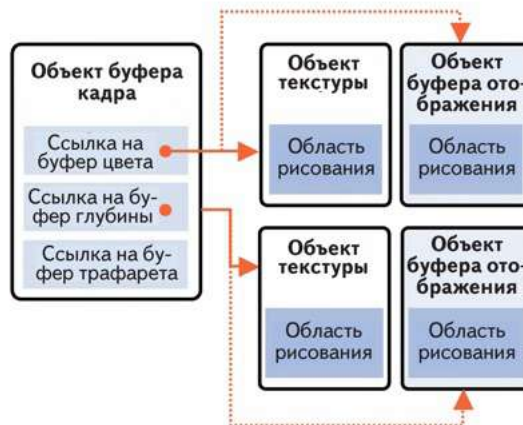


Рис. 10.19. Объект буфера кадра, объект текстуры, объект буфера отображения

WebGL поддерживает два типа объектов, позволяющих рисовать внутри них: **объект текстуры**, с которым мы познакомились в главе 5, и **объект буфера отображения** (renderbuffer object). Изображение, нарисованное в объекте текстуры, можно использовать в качестве текстуры. Объект буфера отображения является более универсальным и позволяет записывать в него самые разные данные.

Как реализовать использование нарисованного объекта в качестве текстуры

Когда возникает необходимость нарисовать в буфере кадра изображение и использовать его в качестве текстуры, для создания этого изображения нужно использовать объект текстуры, подключенный как буфер цвета. Так как при этом часто требуется удалить скрытые поверхности, объект буфера кадра следует настроить, как показано на рис. 10.20.

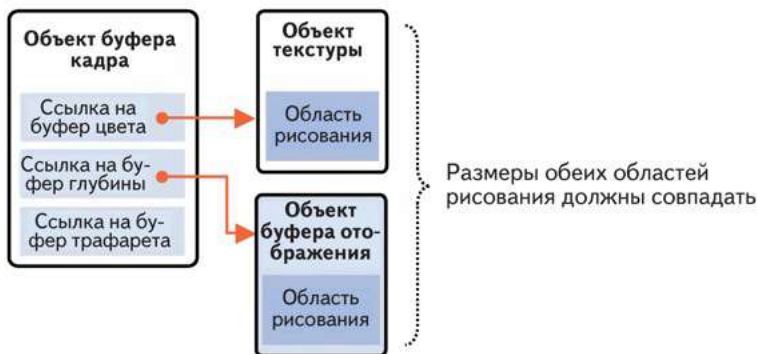


Рис. 10.20. Конфигурация объекта буфера кадра при использовании нарисованного в нем изображения в качестве текстуры

Для этого следует выполнить следующие восемь шагов. Эти шаги напоминают процесс настройки типичного объекта буфера. Шаг 2 описан в главе 5, то есть новыми для нас являются семь из восьми:

1. Создать объект буфера кадра (`gl.createFramebuffer()`).
2. Создать объект текстуры и задать его размер и параметры (`gl.createTexture()`, `gl.bindTexture()`, `gl.texImage2D()`, `gl.Parameteri()`).
3. Создать объект буфера отображения (`gl.createRenderbuffer()`).
4. Связать объект буфера отображения с его типом и задать размер (`gl.bindRenderbuffer()`, `gl.renderbufferStorage()`).
5. Подключить объект текстуры к объекту буфера кадра как ссылку на буфер цвета (`gl.bindFramebuffer()`, `gl.framebufferTexture2D()`).
6. Подключить буфера отображения к объекту буфера кадра как ссылку на буфер глубины (`gl.framebufferRenderbuffer()`).
7. Проверить корректность настройки объекта буфера кадра (`gl.checkFramebufferStatus()`).
8. Нарисовать изображение в объекте буфера кадра (`gl.bindFramebuffer()`).

Теперь посмотрим, как эти шаги реализованы в программном коде. Числа в скобках справа отмечают номера шагов в списке выше.

Пример программы (FramebufferObject.js)

Шаги с 1 по 7, из 8 реализованных в программе `FramebufferObject.js`, показаны в листинге 10.13.

Листинг 10.13. `FramebufferObject.js` (реализация шагов с 1 по 7)

```

1 // FramebufferObject.js
...
24 // Размеры буфера в памяти
25 var OFFSCREEN_WIDTH = 256;
26 var OFFSCREEN_HEIGHT = 256;
27
28 function main() {
...
55 // Определить информацию о вершинах
56 var cube = initVertexBuffersForCube(gl);
57 var plane = initVertexBuffersForPlane(gl);
...
64 var texture = initTextures(gl);
...
70 // Инициализировать объект буфера кадра (framebuffer object, FBO)
71 var fbo = initFramebufferObject(gl);
...
80 var viewProjMatrix = new Matrix4(); // Для буфера цвета
81 viewProjMatrix.setPerspective(30, canvas.width/canvas.height, 1.0, 100.0);
82 viewProjMatrix.lookAt(0.0, 0.0, 7.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
83
84 var viewProjMatrixFBO = new Matrix4(); // Для объекта буфера кадра
85 viewProjMatrixFBO.setPerspective(30.0, OFFSCREEN_WIDTH/OFFSCREEN_HEIGHT,
    ➤ 1.0, 100.0);
86 viewProjMatrixFBO.lookAt(0.0, 2.0, 7.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
...
92 draw(gl, canvas, fbo, plane, cube, currentAngle, texture, viewProjMatrix,
    ➤ viewProjMatrixFBO);
...
96 }
...
263 function initFramebufferObject(gl) {
264   var framebuffer, texture, depthBuffer;
...
274   // Создать объект буфера кадра (FBO) <-(1)
275   framebuffer = gl.createFramebuffer();
...
281   // Создать объект текстуры, установить его размер и параметры <-(2)
282   texture = gl.createTexture(); // Создать объект текстуры
...
287   gl.bindTexture(gl.TEXTURE_2D, texture);
288   gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, OFFSCREEN_WIDTH,
    ➤ OFFSCREEN_HEIGHT, 0, gl.RGBA, gl.UNSIGNED_BYTE, null);
289   gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
290   framebuffer.texture = texture; // Сохранить ссылку на объект текстуры
291
292   // Создать объект буфера отображения, установить его размер и параметры

```



```
293 depthBuffer = gl.createRenderbuffer(); // Создать буфер отображения <-(3)
    ...
298 gl.bindRenderbuffer(gl.RENDERBUFFER, depthBuffer); <-(4)
299 gl.renderbufferStorage(gl.RENDERBUFFER, gl.DEPTH_COMPONENT16,
    ↳OFFSCREEN_WIDTH, OFFSCREEN_HEIGHT);
300
301 // Подключить объекты текстуры и буфера отображения
302 gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer);
303 gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0,
    ↳gl.TEXTURE_2D, texture, 0); <-(5)
304 gl.framebufferRenderbuffer(gl.FRAMEBUFFER, gl.DEPTH_ATTACHMENT,
    ↳gl.RENDERBUFFER, depthBuffer); <-(6)
305
306 // Проверить корректность настройки FBO <-(7)
307 var e = gl.checkFramebufferStatus(gl.FRAMEBUFFER);
308 if (e !== gl.FRAMEBUFFER_COMPLETE) {
309     console.log('Framebuffer object is incomplete: ' + e.toString());
310     return error();
311 }
312
    ...
319 return framebuffer;
320 }
```

Мы опустили вершинный и фрагментный шейдеры, потому что здесь используются те же шейдеры, что и в программе `TexturedQuad.js` из главы 5, которая накладывает текстуру на прямоугольник. Программа в этом разделе рисует два объекта: куб и прямоугольник. Так же как в программе `ProgramObject.js` из предыдущего раздела, мы создаем по несколько объектов буферов для каждого рисуемого объекта и присваиваем ссылки на них свойствам объектов типа `Object`. Затем сохраняем эти объекты в переменных `cube` и `plane`. Мы будем использовать их для рисования, присваивая каждый буфер соответствующей переменной-атрибуту.

Ключевым местом в этой программе является инициализация объекта буфера кадра в `initFramebufferObject()`, в строке 71. Инициализированный объект буфера кадра сохраняется в переменной `fbo` и передается в третьем аргументе функции `draw()`, в строке 92. К функции `draw()` мы еще вернемся, а пока рассмотрим функцию `initFramebufferObject()`, определение которой начинается в строке 263. Эта функция реализует шаги с 1 по 7. Матрица вида проекции для объекта буфера кадра готовится отдельно (строка 84), потому что она отличается от аналогичной матрицы для буфера цвета.

Создание объекта буфера кадра (`gl.createFramebuffer()`)

Прежде чем использовать объект буфера кадра, его нужно создать. Программа в данном разделе создает этот объект в строке 275:

```
275 framebuffer = gl.createFramebuffer();
```

Для этого она использует функцию `gl.createFramebuffer()`.

`gl.createFramebuffer()`

Создает объект буфера кадра.

Параметры	нет	
Возвращаемое значение	непустое	Созданный объект буфера кадра.
	null	Произошла ошибка.
Ошибки	нет	

Удалить объект буфера кадра можно с помощью функции `gl.deleteFramebuffer()`.

`gl.deleteFramebuffer(framebuffer)`

Удаляет объект буфера кадра.

Параметры	framebuffer	Определяет объект буфера кадра, подлежащий удалению.
Возвращаемое значение	нет	
Ошибки	нет	

После создания объекта буфера кадра к нему необходимо подключить объект текстуры (к ссылке на буфер цвета) и объект буфера отображения (к ссылке на буфер глубины). Начнем с создания объекта текстуры.

Создание объекта текстуры, настройка его размеров и параметров

Мы уже видели в главе 5, как создать объект текстуры и настроить его параметры (`gl.TEXTURE_MIN_FILTER`). Обратите внимание, что его ширина и высота определены в виде переменных `OFFSCREEN_WIDTH` и `OFFSCREEN_HEIGHT`, соответственно. Размеры выбраны меньше, чем размеры элемента `<canvas>`, чтобы ускорить процедуру рисования.

```

282 texture = gl.createTexture(); // Создать объект текстуры
    ...
287 gl.bindTexture(gl.TEXTURE_2D, texture);
288 gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, OFFSCREEN_WIDTH,
    └─OFFSCREEN_HEIGHT, 0, gl.RGBA, gl.UNSIGNED_BYTE, null);
289 gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
290 framebuffer.texture = texture; // Сохранить ссылку на объект текстуры

```

Функция `gl.texImage2D()`, что вызывается в строке 288, создает область рисования в объекте текстуры. В последнем аргументе мы передаем этой функции значение `null`; этот аргумент используется для передачи ссылки на объект `Image`.

Объект текстуры нам еще понадобится позже, поэтому далее мы сохраняем его в свойстве `framebuffer.texture`, в строке 290.

На этом подготовка буфера текстуры заканчивается. Далее нам нужно создать объект буфера отображения, который будет играть роль буфера глубины.

Создание объекта буфера отображения (`gl.createRenderbuffer()`)

Как и в случае с объектом текстуры, нам требуется создать объект буфера отображения, прежде чем использовать его. В программе эта операция выполняется в строке 293.

```
293 depthBuffer = gl.createRenderbuffer(); // Создать буфер отображения
```

С этой целью вызывается функция `gl.createRenderbuffer()`.

<code>gl.createRenderbuffer()</code>		
Создает объект буфера отображения.		
Параметры	нет	
Возвращаемое значение	непустое	Созданный объект буфера отображения.
	<code>null</code>	Произошла ошибка.
Ошибки	нет	

Удалить объект буфера отображения можно с помощью функции `gl.deleteRenderbuffer()`.

<code>gl.deleteRenderbuffer(renderbuffer)</code>		
Удаляет объект буфера отображения.		
Параметры	<code>renderbuffer</code>	Определяет объект буфера отображения, подлежащий удалению.
Возвращаемое значение	нет	
Ошибки	нет	

Созданный объект буфера отображения будет использоваться далее в роли буфера глубины, поэтому мы сохраняем его в переменной `depthBuffer`.

Связывание объекта буфера отображения с типом и настройка его размера (`gl.bindRenderbuffer()`, `gl.renderbufferStorage()`)

Прежде чем использовать созданный объект буфера отображения, нужно связать этот объект с его типом и затем выполнять операции с этой целью.

```

298 gl.bindRenderbuffer(gl.RENDERBUFFER, depthBuffer);
299 gl.renderbufferStorage(gl.RENDERBUFFER, gl.DEPTH_COMPONENT16,
    ➔OFFSCREEN_WIDTH, OFFSCREEN_HEIGHT);

```

Связывание объекта буфера отображения с его типом осуществляется вызовом функции `gl.bindRenderbuffer()`.

gl.bindRenderbuffer(target, renderbuffer)

Определяет тип `target` объекта буфера отображения `renderbuffer`. Если в аргументе `renderbuffer` передать `null`, произойдет разрыв связи объекта `renderbuffer` с типом `target`.

Параметры	<code>target</code>	Должен иметь значение <code>gl.RENDERBUFFER</code> .
	<code>renderbuffer</code>	Объект буфера отображения, тип которого требуется указать.
Возвращаемое значение	нет	
Ошибки	<code>INVALID_ENUM</code>	В параметре <code>target</code> передано значение, отличное от <code>gl.RENDERBUFFER</code> .

После связывания объекта буфера отображения с его типом требуется определить его формат, ширину и высоту, вызвав функцию `gl.renderbufferStorage()`. При этом ширина и высота должны совпадать с шириной и высотой объекта текстуры, используемого в роли буфера цвета.

gl.renderbufferStorage(target, internalformat, width, height)

Создает и инициализирует хранилище данных для объекта буфера отображения.

Параметры	<code>target</code>	Должен иметь значение <code>gl.RENDERBUFFER</code> .
	<code>internalformat</code>	Формат буфера отображения.
	<code>gl.DEPTH_COMPONENT16</code>	Буфер отображения используется как буфер глубины.
	<code>gl.STENCIL_INDEX8</code>	Буфер отображения используется как буфер трафарета.
	<code>gl.RGBA4</code>	Буфер отображения используется как буфер цвета. Когда указано значение <code>gl.RGBA4</code> , для каждого компонента цвета <code>RGBA</code> отводится по 4 бита. Когда указано значение <code>gl.RGB5_A1</code> , для каждого компонента цвета <code>RGB</code> отводится по 5 бит, а для компонента <code>A</code> – один бит. Когда указано значение <code>gl.RGB565</code> , для каждого компонента цвета <code>RGB</code> отводится по 5, 6 и 5 бит, соответственно.
	<code>gl.RGB5_A1</code>	
	<code>gl.RGB565</code>	
	<code>width, height</code>	Определяют ширину и высоту буфера отображения в пикселях.
Возвращаемое значение	нет	

Ошибки	INVALID_ENUM	В параметре <code>target</code> передано значение, отличное от <code>gl.RENDERBUFFER</code> , или в параметре <code>internalformat</code> передано недопустимое значение, не являющееся одним из перечисленных.
	INVALID_OPERATION	Нет буфера отображения, связанного с указанным типом <code>target</code> .

На этом подготовку объектов текстуры и буфера отображения можно считать законченной. На данном этапе мы уже можем использовать созданные объекты для рисования в памяти.

Подключение объекта текстуры, как ссылки на буфер цвета (`gl.bindFramebuffer()`, `gl.framebufferTexture2D()`)

Дальнейшая подготовка объекта буфера кадра выполняется так же, как объекта буфера отображения: нужно связать его с типом и затем манипулировать уже этим типом, а не с самим объектом буфера кадра.

```
302 gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer);
303 gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0,
    ➤ gl.TEXTURE_2D, texture, 0);
```

Связывание объекта буфера кадра с его типом выполняется вызовом `gl.bindFramebuffer()`.

<code>gl.bindFramebuffer(target, framebuffer)</code>		
Определяет тип <code>target</code> объекта буфера кадра <code>framebuffer</code> . Если в аргументе <code>framebuffer</code> передать <code>null</code> , связь будет разорвана.		
Параметры	<code>target</code>	Должен иметь значение <code>gl.FRAMEBUFFER</code> .
	<code>framebuffer</code>	Объект буфера кадра.
Возвращаемое значение	нет	
Ошибки	INVALID_ENUM	В параметре <code>target</code> передано значение, отличное от <code>gl.FRAMEBUFFER</code> .

После связывания объекта буфера кадра с его типом, можно сохранить ссылку на объект текстуры в объекте буфера кадра. В нашем примере мы используем объект текстуры вместо буфера цвета, поэтому его нужно подключить к объекту буфера кадра, как ссылку на буфер цвета. Сделать это можно вызовом `gl.framebufferTexture2D()`.

<code>gl.framebufferTexture2D(target, attachment, textarget, texture, level)</code>
Подключает объект текстуры <code>texture</code> к объекту буфера кадра с типом <code>target</code> .

Параметры	<code>target</code>	Должен иметь значение <code>gl.FRAMEBUFFER</code> .
	<code>attachment</code>	Определяет точку подключения к буферу кадра.
	<code>gl.COLOR_ATTACHMENT0</code>	Объект <code>texture</code> используется как буфер цвета.
	<code>gl.DEPTH_ATTACHMENT</code>	Объект <code>texture</code> используется как буфер глубины.
	<code>textarget</code>	Определяет первый аргумент в вызове <code>gl.texImage2D()</code> (<code>gl.TEXTURE_2D</code> или <code>gl.CUBE_MAP_TEXTURE</code>).
	<code>texture</code>	Определяет объект текстуры для подключения к объекту буфера кадра.
	<code>level</code>	Должен иметь значение 0 (если объект текстуры <code>texture</code> предусматривает возможность MIP-текстурирования, в аргументе <code>level</code> нужно указать уровень детализации).
Возвращаемое значение	нет	
Ошибки	<code>INVALID_ENUM</code>	В параметре <code>target</code> передано значение, отличное от <code>gl.FRAMEBUFFER</code> . В параметре <code>attachment</code> или <code>textarget</code> передано значение, не являющееся одним из перечисленных выше.
	<code>INVALID_VALUE</code>	Параметр <code>level</code> имеет недопустимое значение.
	<code>INVALID_OPERATION</code>	Нет буфера кадра, связанного с указанным типом <code>target</code> .

Наличие цифры 0 в имени константы `gl.COLOR_ATTACHMENT0`, которая передается в параметре `attachment`, объясняется тем, что объект буфера кадра в OpenGL, лежащей в основе WebGL, может иметь ссылки на несколько буферов цвета (`gl.COLOR_ATTACHMENT0`, `gl.COLOR_ATTACHMENT1`, `gl.COLOR_ATTACHMENT2` ...). Однако в WebGL поддерживается только одна ссылка.

После подключения буфера текстуры на место буфера цвета, нужно точно так же подключить объект буфера отображения на место буфера глубины. Процедура подключения выполняется аналогично.

Подключение объекта буфера отображения к объекту буфера кадра (`gl.framebufferRenderbuffer()`)

Подключение объекта буфера отображения к объекту буфера кадра осуществляется вызовом `gl.framebufferRenderbuffer()`. Нам нужен буфер глубины, чтобы обеспечить удаление скрытых поверхностей. Поэтому буфер отображения подключается как ссылка на буфер глубины.

```
304 gl.framebufferRenderbuffer(gl.FRAMEBUFFER, gl.DEPTH_ATTACHMENT,
    ➤ gl.RENDERBUFFER, depthBuffer);
```

`gl.framebufferRenderbuffer(target, attachment, renderbuffertarget, renderbuffer)`

Подключает объект буфера отображения `renderbuffertarget` к объекту буфера кадра, связанному с типом `target`.

Параметры	<code>target</code>	Должен иметь значение <code>gl.FRAMEBUFFER</code> .
	<code>attachment</code>	Определяет точку подключения к объекту буфера кадра
	<code>gl.COLOR_ATTACHMENT0</code>	<code>renderbuffer</code> используется как буфер цвета.
	<code>gl.DEPTH_ATTACHMENT</code>	<code>renderbuffer</code> используется как буфер глубины.
	<code>gl.STENCIL_ATTACHMENT</code>	<code>renderbuffer</code> используется как буфер трафарета.
	<code>renderbuffertarget</code>	Должен иметь значение <code>gl.RENDERBUFFER</code>
	<code>renderbuffer</code>	Объект буфера отображения, подключаемый к объекту буфера кадра.

Возвращаемое значение нет

Ошибки `INVALID_ENUM`

В параметре `target` передано значение, отличное от `gl.FRAMEBUFFER`, или в параметре `attachment` передано недопустимое значение, не являющееся одним из перечисленных выше, или в параметре `renderbuffertarget` передано значение, отличное от `gl.RENDERBUFFER`.

Теперь, завершив подготовку буферов цвета и глубины в объекте буфера кадра, можно приступить к рисованию. Но перед этим желательно убедиться, что все настройки выполнены правильно.

Проверка корректности настройки объекта буфера кадра (`gl.checkFramebufferStatus()`)

Очевидно, что попытки использовать неправильно настроенный объект буфера кадра будут приводить к ошибкам. Как вы могли видеть в нескольких последних разделах, подготовка объектов текстуры и буфера отображения представляет собой достаточно сложную процедуру, при реализации которой легко допустить ошибку. Проверить корректность создания и настройки объекта буфера кадра можно с помощью `gl.checkFramebufferStatus()`.

```

307  var e = gl.checkFramebufferStatus(gl.FRAMEBUFFER);
308  if (e !== gl.FRAMEBUFFER_COMPLETE) {
309      console.log('Framebuffer object is incomplete: ' + e.toString());
310      return error();
311  }
```


gl.checkFramebufferStatus(target)

Проверяет состояние готовности буфера кадра, связанного с типом target.

Параметры	target	Должен иметь значение gl.FRAMEBUFFER.
Возвращаемое значение	0	В параметре target передано значение, отличное от gl.FRAMEBUFFER.
	Другие значения	
	gl.FRAMEBUFFER_COMPLETE	Объект буфера кадра настроен правильно.
	gl.FRAMEBUFFER_INCOMPLETE_ATTACHMENT	Настройка точек подключения выполнена не до конца. (Подключенные объекты не готовы к использованию. Один или оба объекта настроены с ошибками.)
	gl.FRAMEBUFFER_INCOMPLETE_DIMENSIONS	Объекты текстуры и буфера отображения имеют разную ширину и/или высоту.
	gl.FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT	Отсутствует ссылка на объект текстуры и/или объект буфера отображения
Ошибки	INVALID_ENUM	В параметре target передано значение, отличное от gl.FRAMEBUFFER.

На этом подготовка объекта буфера кадра закончена полностью. Теперь перейдем к функции draw().

Рисование с использованием объекта буфера кадра

В листинге 10.14 приводится исходный код функции draw(). Она переключает механизм рисования на объект fbo (объект буфера кадра) и рисует куб в объекте текстуры. Затем эта текстура передается функции drawTexturedPlane(), которая рисует прямоугольник в буфере цвета.

Листинг 10.14. FramebufferObject.js (шар (8))

```

321 function draw(gl, canvas, fbo, plane, cube, angle, texture, viewProjMatrix,
    ↪viewProjMatrixFBO) {
322   gl.bindFramebuffer(gl.FRAMEBUFFER, fbo);
323   gl.viewport(0, 0, OFFSCREEN_WIDTH, OFFSCREEN_HEIGHT); // На буфер кадра
324
325   gl.clearColor(0.2, 0.2, 0.4, 1.0); // Немного изменить цвет
326   gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT); // Очистить буфер
327   // Нарисовать
328   drawTexturedCube(gl, gl.program, cube, angle, texture, viewProjMatrixFBO);
329   // Переключиться на буфер цвета
330   gl.bindFramebuffer(gl.FRAMEBUFFER, null);
331   // Восстановить прежние размеры области рисования
332   gl.viewport(0, 0, canvas.width, canvas.height);
333   gl.clearColor(0.0, 0.0, 0.0, 1.0);
334   gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
335   // Нарисовать прямоугольник

```



```
336 drawTexturedPlane(gl, gl.program, plane, angle, fbo.texture, viewProjMatrix);
337 }
```

Строка 322 переключает механизм рисования на использование объекта буфера кадра вызовом `gl.bindFramebuffer()`. После этого все операции рисования с применением `gl.drawArrays()` или `gl.drawElements()` будут выполняться в объекте буфера кадра. В строке 323 вызывается функция `gl.viewport()`, чтобы указать размеры области рисования в буфере (то есть, в памяти).

`gl.viewport(x, y, width, height)`

Определяет область рисования для функций `gl.drawArrays()` и `gl.drawElements()`. В WebGL параметры `x` и `y` задаются в системе координат элемента `<canvas>`.

Параметры `x, y` Координаты нижнего левого угла прямоугольника области рисования (в пикселях).

`width, height` Ширина и высота области рисования (в пикселях).

Возвращаемое значение нет

Ошибки нет

Строка 326 очищает объекты текстуры и буфера глубины, связанные с объектом буфера кадра. Когда в строке 328 вызывается функция рисования куба, изображение получается в объекте текстуры. Чтобы результат был заметнее на общем черном фоне, в строке 325 задается немного другой цвет очистки – пурпурный с багряным оттенком. В итоге в объекте текстуры создается изображение куба, которое можно использовать в качестве текстуры. Далее нам нужно нарисовать прямоугольник и наложить на него только что нарисованную текстуру. В этом случае, из-за того, что рисование должно осуществляться в буфере цвета, нужно переключить вывод обратно в буфер цвета. Эта операция выполняется в строке 330, за счет передачи функции `gl.bindFramebuffer()` значения `null` во втором аргументе (таким способом мы разрываем связь). Затем, в строке 336, осуществляется рисование прямоугольника. Обратите внимание, что в аргументе `texture` функции `drawTexturedPlane()` передается объект текстуры `fbo.texture`. Наиболее внимательные читатели могли заметить, что в этой программе текстура накладывается также на обратную сторону прямоугольника. Это обусловлено тем, что по умолчанию WebGL рисует обе стороны многоугольников. Вы можете отменить рисование обратной стороны, включив механизм **отбраковки**, вызовом `gl.enable(gl.CULL_FACE)`, что даст увеличение скорости рисования (в идеале – в два раза).

Отображение теней

В главе 8 рассказывалось о затенении, одном из явлений, связанных с освещением объектов. Мы коротко упоминали так же и о тенях – еще одном явлении, но не показывали, как реализовать его. Сейчас мы собираемся восполнить этот пробел.

Существует множество способов реализации теней, но мы остановимся лишь на одном из них, который основан на использовании карты теней (карты глубин). Это весьма выразительный метод и широко применяется не только в компьютерной графике, но и для создания специальных эффектов в фильмах.

Как реализуются тени

Метод карты теней основан на идее, что солнце не может «видеть» тени объектов. Суть метода в том, чтобы поместить точку наблюдения в позицию источника света и посмотреть, что можно видеть с этой точки. Все видимые объекты должны быть освещенными. Все, что находится позади этих объектов, должно быть в тени. В реализации этого метода можно использовать расстояние от источника света до объектов (фактически, мы будем использовать z -значение, то есть значение глубины), чтобы определить, какие объекты видимы. На рис. 10.21, где показаны две точки на одной линии, $p1$ и $p2$, точка $p2$ находится в тени, потому что расстояние от источника света до точки $p2$ больше, чем расстояние до точки $p1$.

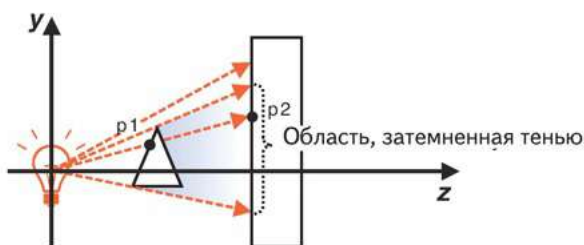


Рис. 10.21. Метод карты теней

Для реализации теней нам потребуется две пары шейдеров: (1) пара шейдеров, которая вычисляет расстояние от источника света до объектов, и (2) пара шейдеров, которая рисует тени с использованием вычисленных расстояний. Нам также понадобится некоторый способ передачи информации о расстоянии от источника света, вычисленном первой парой шейдеров, во вторую пару шейдеров. С этой целью можно использовать изображение текстуры. Такое изображение называют **картой теней**, а метод, основанный на создании карты теней, называют **наложением теней**. Прием наложения теней выполняется в два этапа:

1. Переместить точку наблюдения в позицию источника света и нарисовать объекты, видимые оттуда. Сохранить в изображении текстуры (карте теней) расстояния от источника света (точки наблюдения) до каждого фрагмента, видимого с этой позиции.
2. Переместить точку наблюдения в исходную позицию и вновь нарисовать объекты. Сравнить расстояние от источника света до фрагмента, нарисованного на этом шаге, с расстоянием, записанным в карту теней на предыдущем шаге. Если первое расстояние больше, фрагмент можно нарисовать, как находящийся в тени (более темным цветом).

Для реализации первого шага мы будем использовать объект буфера кадра, и сохранять в нем расстояния. Соответственно, объект буфера кадра в данном случае будет настроен точно так же, как в программе `FramebufferObject.js` (см. рис. 10.20). Нам так же потребуется реализовать переключение между парами шейдеров, при переходе от шага 1 к шагу 2, используя прием, описанный в разделе «Переключение шейдеров», выше в этой главе.

Теперь перейдем к примеру программы `Shadow`. На рис. 10.22 показан скриншот этой программы, где можно видеть тень от красного треугольника, отбрасываемую на наклоненный белый прямоугольник.

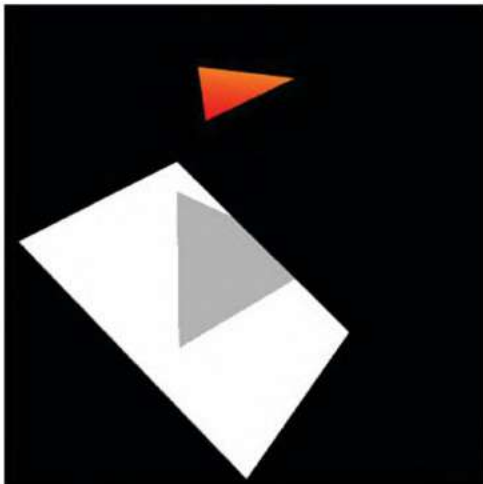


Рис. 10.22. `Shadow`

Пример программы (`Shadow.js`)

Ключевую роль в реализации теней играют шейдеры, представленные в листинге 10.15.

Листинг 10.15. `Shadow.js` (шейдеры)

```
1 // Shadow.js
2 // Вершинный шейдер, генерирующий карту теней
3 var SHADOW_VSHADER_SOURCE =
4     ...
5
6     'void main() {\n' +
7     '   gl_Position = u_MvpMatrix * a_Position;\n' +
8     '}\n';
9
10 // Фрагментный шейдер, генерирующий карту теней
11 var SHADOW_FSHADER_SOURCE =
12     ...
13
14     'void main() {\n' +
15     '   gl_FragColor = vec4(gl_FragCoord.z, 0.0, 0.0, 0.0);\n' +
16     '}\n';
17
18
19 // Вершинный шейдер для обычного рисования
20 var VSHADER_SOURCE =
21     ...
22
23     'uniform mat4 u_MvpMatrix;\n' +
24     'uniform mat4 u_MvpMatrixFromLight;\n' +
25     'varying vec4 v_PositionFromLight;\n' +
26     'varying vec4 v_Color;\n' +
27     'void main() {\n' +
28     '   gl_Position = u_MvpMatrix * a_Position;\n' +
```



```

29 '   v_PositionFromLight = u_MvpMatrixFromLight * a_Position;\n' +
30 '   v_Color = a_Color;\n' +
31 '}\n';
32
33 // Фрагментный шейдер для обычного рисования
34 var FSHADER_SOURCE =
35     ...
38 'uniform sampler2D u_ShadowMap;\n' +
39 'varying vec4 v_PositionFromLight;\n' +
40 'varying vec4 v_Color;\n' +
41 'void main() {\n' +
42 '   vec3 shadowCoord = (v_PositionFromLight.xyz/v_PositionFromLight.w)
43                       * 2.0 + 0.5;\n' +
44 '   vec4 rgbaDepth = texture2D(u_ShadowMap, shadowCoord.xy);\n' +
45 '   float depth = rgbaDepth.r;\n' + // Получить z-значение из R
46 '   float visibility = (shadowCoord.z > depth + 0.005) ? 0.7:1.0;\n' + <-(2)
47 '   gl_FragColor = vec4(v_Color.rgb * visibility, v_Color.a);\n' +
48 '}\n';

```

Шаг 1 выполняется парой шейдеров, отвечающих за создание карты теней и реализованных в строках с 3 по 17. Программе на JavaScript остается только переключить рисование в объект буфера кадра, передать матрицу модели вида проекции для точки наблюдения, находящейся в позиции источника света, в переменную `u_MvpMatrix` и нарисовать объекты. В результате расстояния от источника света до фрагментов будут записаны в текстуру (карту теней), подключенную к объекту буфера кадра. Вершинный шейдер в строке 7 просто умножает матрицу модели вида проекции на координаты вершин, чтобы получить расстояние. Фрагментный шейдер устроен сложнее и в нем требуется найти расстояние от источника света до рисуемого фрагмента. С этой целью можно использовать встроенную переменную `gl_FragCoord`, описанную в главе 5.

`gl_FragCoord` – это встроенная переменная типа `vec4`, хранящая координаты текущего фрагмента. Компоненты `gl_FragCoord.x` и `gl_FragCoord.y` представляют позицию фрагмента на экране, а компонент `gl_FragCoord.z` содержит нормализованное *z*-значение в диапазоне $[0, 1]$. Оно вычисляется по формуле: $(gl_Position.z/gl_Position.w)/2.0+0.5$. (Подробности см. в разделе 2.12 спецификации OpenGL ES 2.0.) Значение компонента `gl_FragCoord.z` находится в диапазоне от 0.0 до 1.0, где значение 0.0 представляет фрагменты, находящиеся на расстоянии ближней плоскости отсечения, а 1.0 – на расстоянии дальней плоскости отсечения. Это значение записывается в компонент *R* (red) цвета (с равным успехом можно использовать любой другой компонент) пикселя в карте теней (строка 16).

```

16 '   gl_FragColor = vec4(gl_FragCoord.z, 0.0, 0.0, 0.0);\n' +           <-(1)

```

Так *z*-значение каждого фрагмента, нарисованного при наблюдении с позиции, совпадающей с позицией источника света, записывается в карту теней. Эта карта теней затем будет передаваться в переменной `u_ShadowMap`, объявленной в строке 38.

Чтобы выполнить шаг 2, нужно нарисовать объекты еще раз, после переключения механизма рисования на использование буфера цвета и переноса точки наблюдения в исходную позицию. После рисования объектов выбирается цвет каждого фрагмента, путем сравнения z-значения фрагмента с тем, что хранится в карте теней. Делается это в обычных шейдерах, реализованных в строках с 20 по 47. В переменной `u_MvpMatrix` передается матрица модели вида проекции для точки наблюдения в исходной позиции, а в переменной `u_MvpMatrixFromLight`, использовавшейся для создания карты теней – матрица модели вида проекции для точки наблюдения в позиции источника света. Главная задача вершинного шейдера, начинающегося в строке 20, заключается в вычислении координат каждого фрагмента, при взгляде с позиции источника света, и передаче их фрагментному шейдеру (строка 29), чтобы можно было получить z-значение для каждого фрагмента относительно источника света.

Фрагментный шейдер использует координаты для вычисления z-значения (строка 42). Как уже упоминалось, карта теней содержит значения $(gl_Position.z/gl_Position.w)/2.0+0.5$. Поэтому нам остается только вычислить z-значение для сравнения со значением в карте теней: $(v_PositionFromLight.z/v_PositionFromLight.w)/2.0+0.5$. Однако, чтобы получить значение текселя из карты теней, в строке 42 выполняются дополнительные вычисления. Чтобы выполнить сравнение со значением из карты теней, нужно получить значение текселя из карты теней, координаты которого соответствуют координатам $(v_PositionFromLight.x, v_PositionFromLight.y)$. Как известно, `v_PositionFromLight.x` и `v_PositionFromLight.y` – это координаты X и Y в системе координат WebGL (см. рис. 2.18 в главе 2), изменяющиеся в диапазоне от -1.0 до 1.0. С другой стороны, координаты текстуры S и T в карте теней изменяются в диапазоне от 0.0 до 1.0 (см. рис. 5.20 в главе 5). Поэтому координаты X и Y следует преобразовать в координаты S и T. Сделать это можно, применив то же выражение, с помощью которого вычислялось z-значение:

- Координата S-текстуры:

$$(v_PositionFromLight.x/v_PositionFromLight.w)/2.0 + 0.5.$$

- Координата T-текстуры:

$$(v_PositionFromLight.y/v_PositionFromLight.w)/2.0 + 0.5.$$

Дополнительные подробности, касающиеся этих вычислений, можно найти в разделе 2.12 спецификации OpenGL ES 2.0⁴. Вычисление обоих значений можно реализовать в одной строке программного кода, как можно видеть в строке 42:

```
42   '   vec3 shadowCoord =(v_PositionFromLight.xyz/v_PositionFromLight.w)
      '   ──/ 2.0 + 0.5;\n' +
43   '   vec4 rgbaDepth = texture2D(u_ShadowMap, shadowCoord.xy);\n' +
44   '   float depth = rgbaDepth.r;\n' + // Получить z-значение из R
```

В строках 43 и 44 шейдер извлекает значение из карты теней. В данном случае извлекается только значение R, обращением к `rgbaDepth.r` в строке 44, потому что

⁴ www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf

мы записывали это значение именно в компонент R (строка 16). Строка 45 проверяет, находится ли фрагмент в тени. Когда позиция фрагмента оказывается больше, чем его глубина (то есть, `shadowCoord.z > depth`), в переменную `visibility` записывается значение 0.7. Далее эта переменная используется в строке 46, чтобы нарисовать тень более темным цветом:

```
45 ' float visibility = (shadowCoord.z > depth + 0.005) ? 0.7:1.0;\n'+
46 ' gl_FragColor = vec4(v_Color.rgb * visibility, v_Color.a);\n' +
```

В строке 45 к значению глубины добавляется небольшое смещение (0.005). Чтобы понять, зачем это нужно, попробуйте запустить программу, удалив это число. Вы увидите полосы, как показано на рис. 10.23. Этот эффект называется **полосы Маха** (Mach band).

Небольшое смещение (0.005) подавляет этот эффект, возникающий из-за ограниченной точности чисел, которые можно хранить в компонентах RGBA. Это довольно сложная проблема, но она стоит того, чтобы разобраться с ней, потому что ей подвержены все аспекты трехмерной графики. Z-значение сохраняется в карте теней в компоненте R значения цвета RGBA, который является 8-битным числом. Это означает, что точность значения R ниже, чем сравниваемого с ним значения `shadowCoord.z`, имеющего тип `float`. Например, пусть z-значение равно 0.1234567. Если представить это значение, задействовав 8 бит (другими словами, имея всего 256 возможных значений), мы не сможем обеспечить точность лучше, чем $1/256$ ($=0.0390625$). То есть, число 0.1234567 можно представить, как показано ниже:

$$0.1234567 / (1 / 256) = 31.6049152$$

Цифры правее десятичной точки нельзя сохранить в 8-битном числе, то есть мы можем сохранить только число 31. Далее, разделив 31 на 256, мы получим число 0.12109375, которое, как видите, меньше первоначального значения (0.1234567). Это означает, что даже если фрагмент находится в той же позиции, его z-значение, хранящееся в карте теней, будет меньше его z-значения в `shadowCoord.z`, что и приводит к появлению полос Маха. Так как проблема возникает из-за того, что точность представления значения R составляет $1/256$ ($=0.00390625$), добавляя небольшое смещение, такое как 0.005, к значению R, можно предотвратить появление полос. Обратите внимание, что можно использовать любое другое смещение больше $1/256$; число 0.005 было выбрано потому, что оно чуть больше $1/256$.



Рис. 10.23. Полосы Маха

Теперь рассмотрим программный код на JavaScript, который передает данные в шейдер (см. листинг 10.16), сосредоточив внимание на матрицах преобразований. Чтобы нарисовать тень, размер текстуры для рисования в памяти объявляется больше размеров <canvas>.

Листинг 10.16. Shadow.js (часть программы на JavaScript)

```

49 var OFFSCREEN_WIDTH = 1024, OFFSCREEN_HEIGHT = 1024;
50 var LIGHT_X = 0, LIGHT_Y = 7, LIGHT_Z = 2;
51
52 function main() {
    ...
63 // Инициализировать шейдеры, генерирующие карту теней
64 var shadowProgram = createProgram(gl, SHADOW_VSHADER_SOURCE,
    ↪SHADOW_FSHADER_SOURCE);
    ...
72 // Инициализировать шейдеры для обычного рисования
73 var normalProgram = createProgram(gl, VSHADER_SOURCE, FSHADER_SOURCE);
    ...
85 // Определить информацию о вершинах
86 var triangle = initVertexBuffersForTriangle(gl);
87 var plane = initVertexBuffersForPlane(gl);
    ...
93 // Инициализировать объект буфера кадра (FBO)
94 var fbo = initFramebufferObject(gl);
    ...
99 gl.activeTexture(gl.TEXTURE0); // Установить объект текстуры в слот
100 gl.bindTexture(gl.TEXTURE_2D, fbo.texture);
    ...
106 var viewProjMatrixFromLight = new Matrix4(); // Для карты теней
107 viewProjMatrixFromLight.setPerspective(70.0,
    ↪OFFSCREEN_WIDTH/OFFSCREEN_HEIGHT, 1.0, 100.0);
108 viewProjMatrixFromLight.lookAt(LIGHT_X, LIGHT_Y, LIGHT_Z, 0.0, 0.0, 0.0,
    ↪1.0, 0.0);
109
110 var viewProjMatrix = new Matrix4(); // Для обычного рисования
111 viewProjMatrix.setPerspective(45, canvas.width/canvas.height, 1.0, 100.0);
112 viewProjMatrix.lookAt(0.0, 7.0, 9.0, 0.0, 0.0, 0.0, 1.0, 0.0);
113
114 var currentAngle = 0.0; // Текущий угол поворота [в градусах]
115 var mvpMatrixFromLight_t = new Matrix4(); // Для треугольника
116 var mvpMatrixFromLight_p = new Matrix4(); // Для прямоугольника
117 var tick = function() {
118     currentAngle = animate(currentAngle);
119     // Переключиться на объект буфера кадра
120     gl.bindFramebuffer(gl.FRAMEBUFFER, fbo);
    ...
124 gl.useProgram(shadowProgram); // Для создания карты теней
125 // Нарисовать треугольник и прямоугольник (карта теней)
126 drawTriangle(gl, shadowProgram, triangle, currentAngle,
    ↪viewProjMatrixFromLight);
127 mvpMatrixFromLight_t.set(g_mvpMatrix); // Используется далее
128 drawPlane(gl, shadowProgram, plane, viewProjMatrixFromLight);

```



```
129     mvpMatrixFromLight_p.set(g_mvpMatrix); // Используется далее
130     // Переключиться на буфер цвета
131     gl.bindFramebuffer(gl.FRAMEBUFFER, null);
132     ...
135     gl.useProgram(normalProgram); // Для обычного рисования
136     gl.uniform1i(normalProgram.u_ShadowMap, 0); // Передать gl.TEXTURE0
137     // Нарисовать треугольник и прямоугольник
138     gl.uniformMatrix4fv(normalProgram.u_MvpMatrixFromLight, false,
139                         ↪mvpMatrixFromLight_t.elements);
139     drawTriangle(gl, normalProgram, triangle, currentAngle, viewProjMatrix);
140     gl.uniformMatrix4fv(normalProgram.u_MvpMatrixFromLight, false,
141                         ↪mvpMatrixFromLight_p.elements);
141     drawPlane(gl, normalProgram, plane, viewProjMatrix);
142
143     window.requestAnimationFrame(tick, canvas);
144 };
145 tick();
146 }
```

Рассмотрим JavaScript-функцию `main()`, определение которой начинается в строке 52. В строке 64 она инициализирует шейдеры, создающие карту теней. В строке 73 инициализируются шейдеры, используемые для обычного рисования. Строки 86 и 87, определяющие информацию о вершинах и вызов `initFramebufferObject()` в строке 94, – это те же операции, которые мы видели в программе `FramebufferObject.js`. Строка 94 готовит объект буфера кадра, содержащий объект текстур, к созданию карты теней. В строках 99 и 100 активируется текстурный слот 0 и определяется тип текстуры. Этот слот передается через переменную `u_ShadowMap` шейдерам, осуществляющим обычное рисование.

В строках со 106 по 108 готовится матрица вида проекции для использования при создании карты теней. Обратите внимание на первые три аргумента (то есть, местоположение точки наблюдения) в вызове функции в строке 108 – это местоположение источника света. В строках со 110 по 112 готовится матрица вида проекции для точки наблюдения, откуда просматривается сцена.

Наконец, программа рисует треугольник и прямоугольник, используя всю исходную информацию. Прежде всего она генерирует карту теней, для чего в строке 120 переключает механизм рисования на использование объекта буфера кадра. В строках 126 и 128 производится рисование объектов с применением шейдеров, предназначенных для создания карты теней (`shadowProgram`). Обратите внимание, что в строках 127 и 129 сохраняются матрицы модели вида проекции для точки наблюдения, находящейся в позиции источника света. После создания карты теней она используется в коде, который начинается со строки 135 и рисует сцену с тенями. В строке 136 карта передается фрагментному шейдеру. В строках 138 и 140, через `u_MvpMatrixFromLight`, передаются матрицы модели вида проекции, сохраненные в строках 127 и 129 соответственно.

Увеличение точности

Несмотря на то, что сцена с тенью была успешно нарисована, данный пример способен обрабатывать только ситуации, когда источник света находится близко к

объектам. Чтобы понять, о чем идет речь, изменим координаты источника света, как показано ниже:

```
50 var LIGHT_X = 0, LIGHT_Y = 40, LIGHT_Z = 2;
```

Если теперь запустить программу, вы увидите, что тень больше не отображается (рис. 10.24 слева). Очевидно, что тень должна быть и иметь вид, как показано на рис. 10.24 справа.

Причина пропадания тени кроется в увеличении расстояния от объекта до источника света, из-за чего значение `gl_FragCoord.z` не может быть сохранено в компоненте R, в карте теней, потому что он имеет размер всего 8 бит. Самое простое решение этой проблемы состоит в том, чтобы использовать не только компонент R, но все четыре компонента R, G, B и A. Иными словами, нужно сохранить значение в 4 отдельных байтах. Для этого есть специальная процедура, которую мы предлагаем рассмотреть в следующем примере. Он отличается от предыдущего только фрагментным шейдером.

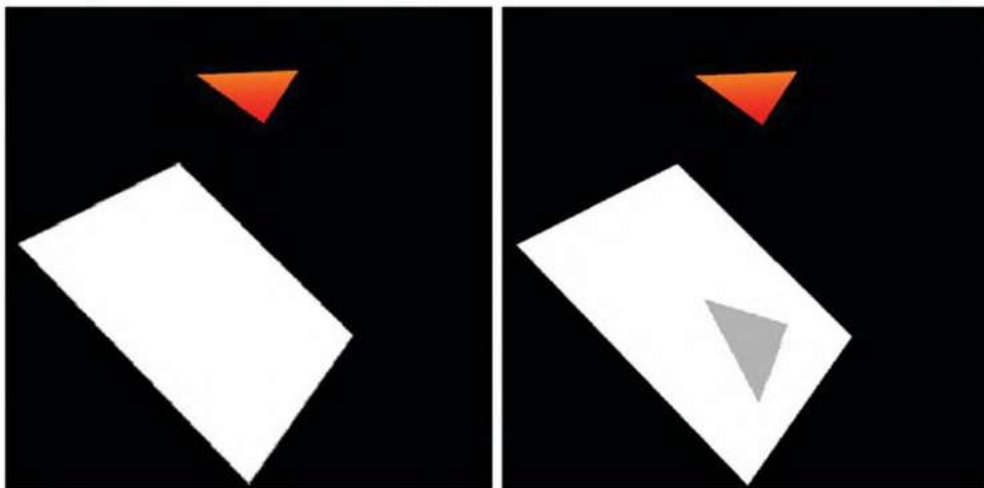


Рис. 10.24. Тень не отображается

Пример программы (*Shadow_highp.js*)

В листинге 10.17 показана реализация фрагментного шейдера из программы `Shadow_highp.js`. В нем можно видеть, что обработка z-значения осложнилась, в сравнении с программой `Shadow.js`.

Листинг 10.17. `Shadow_highp.js`

```
1 // Shadow_highp.js
...
10 // Фрагментный шейдер для создания карты теней
11 var SHADOW_FSHADER_SOURCE =
```

```

...
15 void main() {\n' +
16 '   const vec4 bitShift = vec4(1.0, 256.0, 256.0 * 256.0, 256.0 * 256.0 *
    ↳256.0);\n' +
17 '   const vec4 bitMask = vec4(1.0/256.0, 1.0/256.0, 1.0/256.0, 0.0);\n' +
18 '   vec4 rgbaDepth = fract(gl_FragCoord.z * bitShift);\n' +
19 '   rgbaDepth -= rgbaDepth.gbaa * bitMask;\n' +
20 '   gl_FragColor = rgbaDepth;\n' +
21 '}\n';

...
37 // Фрагментный шейдер для обычного рисования
38 var FSHADER_SOURCE =
    ...
45 // Восстановить z-значение из RGBA
46 'float unpackDepth(const in vec4 rgbaDepth) {\n' +
47 '   const vec4 bitShift = vec4(1.0, 1.0/256.0, 1.0/(256.0 * 256.0),
    ↳1.0/(256.0 * 256.0 * 256.0));\n' +
48 '   float depth = dot(rgbaDepth, bitShift);\n' +
49 '   return depth;\n' +
50 '}\n' +
51 'void main() {\n' +
52 '   vec3 shadowCoord = (v_PositionFromLight.xyz /
    ↳v_PositionFromLight.w)/2.0 + 0.5;\n' +
53 '   vec4 rgbaDepth = texture2D(u_ShadowMap, shadowCoord.xy);\n' +
54 '   float depth = unpackDepth(rgbaDepth);\n' + // Восстановить z-значение
55 '   float visibility = (shadowCoord.z > depth + 0.0015)? 0.7:1.0;\n' +
56 '   gl_FragColor = vec4(v_Color.rgb * visibility, v_Color.a);\n' +
57 '}\n';

```

Код в строках с 16 по 19 разбивает `gl_FragCoord.z` на 4 байта (RGBA). Так как 1 байт может представлять значения с точностью до $1/256$, мы можем хранить в компоненте R только часть значения, которая больше $1/256$. Часть значения меньше $1/256$ и больше $1/(256*256)$ – в G, часть значения меньше $1/(256*256)$ и больше $1/(256*256*256)$ – в B, и остальную часть значения – в A. Строка 18 вычисляет каждую часть значения и сохраняет ее в компонентах RGBA. Вычисления удалось записать в одну строку, благодаря применению типа данных `vec4`. Функция `fract()` – это встроенная функция, которая отбрасывает дробную часть в значении, указанном в виде аргумента. Каждый элемент в `vec4`, что вычисляется в строке 18, имеет точность выше, чем 1 байт, поэтому в строке 19 отбрасываются значения, не укладывающиеся в 1 байт. Присваивая этот результат переменной `gl_FragColor` в строке 20, мы можем сохранить z-значение во всех четырех компонентах RGBA и получить более высокую точность.

Функция `unpackDepth()`, что вызывается в строке 54, читает z-значение из компонентов RGBA. Определение этой функции начинается в строке 46. В строке 48 она выполняет вычисления, связанные с преобразованием значения RGBA в оригинальное z-значение.

$$depth = rgbaDepth.r \times 1.0 + \frac{rgbaDepth.g}{256.0} + \frac{rgbaDepth.b}{(256.0 \times 256.0)} + \frac{rgbaDepth.a}{(256.0 \times 256.0 \times 256.0)}$$

Как видите, вычисления совпадают с вычислением скалярного произведения, поэтому в строке 48 для этого используется функция `dot()`.

Теперь мы можем благополучно извлекать расстояние (*z*-значение) и нам остается всего лишь извлечь расстояние из `shadowCoord.z` (строка 55) и нарисовать тень. В данном случае для устранения эффекта полос Маха используется число 0.0015, вместо 0.005, потому что точность *z*-значения, хранимого в карте теней, имеет тип `float` с точностью `medium` (то есть, $2^{-10} = 0.000976563$, как показано в табл. 6.15). Теперь тень будет отображаться правильно.

Загрузка и отображение трехмерных моделей

В предыдущих главах мы рисовали трехмерные объекты, указывая координаты вершин и информацию о цвете вручную, в виде массивов типа `Float32Array`. Однако, как отмечалось ранее в этой книге, часто бывает нужно прочитать координаты вершин и информацию о цвете из файлов трехмерных моделей, сконструированных с применением инструментов трехмерного моделирования.

В этом разделе мы напишем программу, которая читает файл с определением трехмерной модели, созданный с помощью инструмента трехмерного моделирования. Для этого примера мы использовали Blender 6 – популярный инструмент, для которого имеется бесплатная версия. Инструмент Blender может экспортировать файлы трехмерных моделей в хорошо известном формате OBJ, который является текстовым и доступным для чтения и анализа человеком. OBJ – это формат файлов с определениями геометрических построений, первоначально разработанный в Wavefront Technologies. Это – открытый формат и может использоваться другими производителями инструментов, предназначенных для работы с трехмерной графикой. Вышесказанное означает не только широкое распространение и использование формата, но и наличие большого числа его вариаций. Чтобы упростить реализацию примера, мы сделаем несколько предположений, таких как отсутствие текстур. Тем не менее, пример поможет вам понять, как читать данные моделей, и даст вам основу для дальнейших экспериментов. Программный код примера спроектирован достаточно универсальным, чтобы дать возможность использовать другие текстовые форматы.

Запустите Blender и создайте куб, как показано на рис. 10.25. Окрасьте одну грань куба оранжевым цветом, а другие – красным. Затем экспортируйте модель в файл с именем `cube.obj`. (Пример этого файла можно найти в каталоге `resources`, в загружаемом пакете примеров программ.) Рассмотрим содержимое файла `cube.obj`, который является текстовым файлом и потому может быть открыт в любом текстовом редакторе.

В листинге 10.18 показано содержимое файла `cube.obj`. Мы добавили номера строк, чтобы проще было ссылаться на них в пояснениях.

Листинг 10.18. cube.obj

```

1 #Blender v2.60 (sub 0) OBJFile: ''
2 # www.blender.org
3 mtlllib cube.mtl
4 o Cube
5 v 1.000000 -1.000000 -1.000000
6 v 1.000000 -1.000000 1.000000
7 v -1.000000 -1.000000 1.000000
8 v -1.000000 -1.000000 -1.000000
9 v 1.000000 1.000000 -1.000000
10 v 1.000000 1.000000 1.000001
11 v -1.000000 1.000000 1.000000
12 v -1.000000 1.000000 -1.000000
13 usemtl Material
14 f1234
15 f5876
16 f2673
17 f3784
18 f5148
19 usemtl Material.001
20 f1562

```

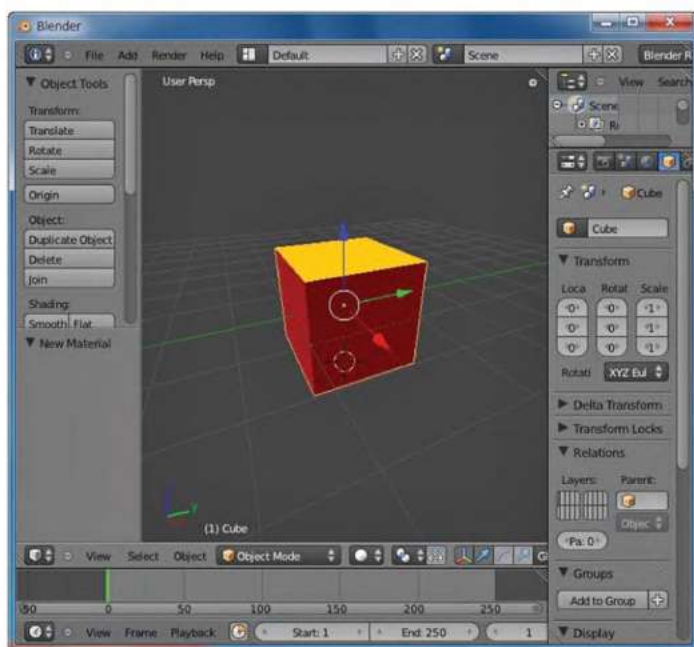


Рис. 10.25. Blender, инструмент трехмерного моделирования

После создания файла модели напомним программу, которая будет читать данные в те же структуры, что использовались нами прежде. Программа должна реализовать следующие шаги:

1. Подготовить массив вершин (`vertices`) типа `Float32Array` и прочитать в него координаты вершин из файла модели.
2. Подготовить массив (`colors`) типа `Float32Array` и прочитать в него цвета из файла модели.
3. Подготовить массив (`normals`) типа `Float32Array` и прочитать в него нормали из файла модели.
4. Подготовить массив (`indices`) типа `Uint16Array` (или `Uint8Array`) и прочитать в него индексы вершин, определяющие треугольники, которые составляют модель, из файла в массив.
5. Записать данные, прочитанные при выполнении шагов с 1 по 4, в буферный объект и нарисовать модель вызовом `gl.drawElements()`.

То есть, в данном случае программа читает данные из файла `cube.obj` (содержимое его показано в листинге 10.18) в соответствующие массивы и затем рисует модель на шаге 5. Чтобы прочесть данные из файла, необходимо знать его формат.

Формат OBJ

Файл в формате OBJ состоит из нескольких разделов,⁵ среди которых разделы с координатами вершин, определениями граней и материалов. В каждом разделе может определяться множество вершин, нормалей и граней:

- строки, начинающиеся с символа решетки (`#`) являются комментариями, то есть строки 1 и 2 в листинге 10.18 – это комментарии, сгенерированные инструментом Blender и содержащие номер версии и адрес проекта в Интернете; остальные строки определяют трехмерную модель;
- строка 3 содержит имя внешнего файла материала; формат OBJ поддерживает хранение информации о материале модели во внешнем файле материала, который обычно называют MTL-файлом, например, строка:

```
mtllib <имя внешнего mtl-файла>
```

указывает, что файл материала имеет имя `cube.mtl`;

- строка 4 определяет имя объекта в следующем формате:

```
<имя объекта>
```

однако в нашей программе эта информация не используется;

- в строках с 5 по 12 определяются координаты вершин в формате `(x,y,z[,w])`, где компонент `w` является необязательным и по умолчанию принимает значение 1.0:

```
v x y z [w]
```

в нашем примере определено восемь вершин, потому что моделью является обычный куб;

⁵ См. http://en.wikipedia.org/wiki/Wavefront_.obj_file (на русском языке: <https://ru.wikipedia.org/wiki/Obj> – Прим. перев.)

- в строках с 13 по 20 определяются материалы и грани, использующие эти материалы; в строке 13 определяется название материала в MTL-файле, на который ссылается строка 4, в формате:

```
usemtl <название материала>
```

- строки с 14 по 18 определяют грани модели из данного материала; грани определяются как списки вершин, текстур и индексов нормалей:

```
f v1 v2 v3 v4 ...
```

где $v1, v2, v3, \dots$ – это индексы вершин, нумерация которых начинается с 1 и соответствует нумерации вершин в списке, определенном выше. Наша программа будет обрабатывать вершины и нормали. В листинге 10.18 нормали не показаны, но, если грань имеет нормаль, используется следующий формат:

```
f v1 // vn1 v2 // vn2 v3 // vn3 ...
```

где $vn1, vn2, vn3, \dots$ – это индексы нормалей, нумерация которых начинается с 1.

Формат файла MTL

Файл MTL может содержать определение нескольких материалов. В листинге 10.19 показано содержимое файла `cube.mtl`.

Листинг 10.19. `cube.mtl`

```
1 # Blender MTL File: ''
2 # Material Count: 2
3 newmtl Material
4 Ka 0.000000 0.000000 0.000000
5 Kd 1.000000 0.000000 0.000000
6 Ks 0.000000 0.000000 0.000000
7 Ns 96.078431
8 Ni 1.000000
9 d 1.000000
10 illum 0
11 newmtl Material.001
12 Ka 0.000000 0.000000 0.000000
13 Kd 1.000000 0.450000 0.000000
14 Ks 0.000000 0.000000 0.000000
15 Ns 96.078431
16 Ni 1.000000
17 d 1.000000
18 illum 0
```

- строки 1 и 2 – это комментарии, сгенерированные инструментом Blender;
- определение каждого нового материала (начиная со строки 3) начинается с команды `newmtl`, определяющей название материала:

`newmtl <название материала>`

которое используется в файле OBJ;

- строки с 4 по 6 определяют цвет отраженного света: фонового (ambient), диффузного (diffuse) и зеркального (specular) в виде команд Ka, Kd и Ks, соответственно; цвет определяется в формате RGB, где каждый компонент имеет значение от 0 до 1. В данной программе используется только цвет диффузного отраженного света;
- строка 7 определяет вес зеркальной составляющей Ns; строка 8 определяет оптическую плотность поверхности в виде параметра Ni; строка 9 определяет прозрачность в виде параметра d; строка 10 определяет яркость освещения модели в виде параметра illum. В данной программе эта часть информации не используется.

Используя только что полученные знания о структуре файлов OBJ и MTL, мы можем реализовать извлечение координат вершин, значений цвета, нормалей и индексов, описывающих грани, записать все это в буферные объекты и нарисовать сцену вызовом `gl.drawElements()`. В файле OBJ может отсутствовать описание нормалей, но их легко вычислить, исходя из координат вершин, используя «векторное произведение».⁶

Перейдем теперь к самой программе.

Пример программы (OBJViewer.js)

Программа выполняет следующие основные шаги: (1) подготовка пустых буферных объектов, (2) чтение файла OBJ (и файла MTL), (3) анализ содержимого, (4) запись результатов в буферные объекты и (5) рисование. Реализация всех этих шагов приводится в листинге 10.20.

Листинг 10.20. OBJViewer.js

```
1 // OBJViewer.js (
...
28 function main() {
...
40 if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
41     console.log('Failed to initialize shaders.');
```

```
42     return;
43 }
...
49 // Получить ссылки на переменные-атрибуты и uniform-переменные
50 var program = gl.program;
51 program.a_Position = gl.getAttribLocation(program, 'a_Position');
52 program.a_Normal = gl.getAttribLocation(program, 'a_Normal');
53 program.a_Color = gl.getAttribLocation(program, 'a_Color');
```

⁶ Если обозначить вершины треугольника как v_0, v_1 и v_2 , вектор v_0 и v_1 как (x_1, y_1, z_1) и вектор v_1 и v_2 как (x_2, y_2, z_2) , тогда векторное произведение можно определить как $(y_1 \cdot z_2 - z_1 \cdot y_2, z_1 \cdot x_2 - x_1 \cdot z_2, x_1 \cdot y_2 - y_1 \cdot x_2)$. Результатом будет нормаль к треугольнику (см. книгу «3D Math Primer for Graphics and Game Development»).

```

    ...
63 // Подготовить пустые буферные объекты
64 var model = initVertexBuffers(gl, program);
    ...
75 // Начать чтение файла OBJ
76 readOBJFile('../resources/cube.obj', gl, model, 60, true);
    ...
81 draw(gl, gl.program, currentAngle, viewProjMatrix, model);
    ...
85 }
86
87 // Создает и настраивает буферный объект
88 function initVertexBuffers(gl, program) {
89     var o = new Object();
90     o.vertexBuffer = createEmptyArrayBuffer(gl, program.a_Position, 3, gl.FLOAT);
91     o.normalBuffer = createEmptyArrayBuffer(gl, program.a_Normal, 3, gl.FLOAT);
92     o.colorBuffer = createEmptyArrayBuffer(gl, program.a_Color, 4, gl.FLOAT);
93     o.indexBuffer = gl.createBuffer();
    ...
98     return o;
99 }
100
101 // Создает буферный объект и присваивает его переменной атрибуту
102 function createEmptyArrayBuffer(gl, a_attribute, num, type) {
103     var buffer = gl.createBuffer(); // Создать буферный объект
    ...
108     gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
109     gl.vertexAttribPointer(a_attribute, num, type, false, 0, 0);
110     gl.enableVertexAttribArray(a_attribute); // Разрешить присваивание
111
112     return buffer;
113 }
114
115 // Читает файл
116 function readOBJFile(fileName, gl, model, scale, reverse) {
117     var request = new XMLHttpRequest();
118
119     request.onreadystatechange = function() {
120         if (request.readyState === 4 && request.status !== 404) {
121             onReadOBJFile(request.responseText, fileName, gl, model, scale, reverse);
122         }
123     }
124     request.open('GET', fileName, true); // Создать запрос, чтобы получить файл
125     request.send(); // Послать запрос
126 }
127
128 var g_objDoc = null; // Информация из файла OBJ
129 var g_drawingInfo = null; // Информация для рисования модели
130
131 // Вызывается для чтения файла OBJ
132 function onReadOBJFile(fileString, fileName, gl, o, scale, reverse) {
133     var objDoc = new OBJDoc(fileName); // Создать объект OBJDoc
134     var result = objDoc.parse(fileString, scale, reverse);
135     if (!result) {

```



```
136     g_objDoc = null; g_drawingInfo = null;
137     console.log("OBJ file parsing error.");
138     return;
139 }
140 g_objDoc = objDoc;
141 }
```

Реализация JavaScript-функции `initVertexBuffers()`, которая вызывается в строке 64, изменилась. Теперь она просто готовит пустые буферные объекты для хранения координат вершин, цветов и нормалей. Заполнение буферных объектов соответствующей информацией будет выполняться после анализа файла OBJ.

В строках с 90 по 92 функция `initVertexBuffers()` (определение начинается в строке 88) создает необходимые буферные объекты и присваивает их переменным-атрибутам, вызывая `createEmptyArrayBuffer()`. Определение функции `createEmptyArrayBuffer()` начинается в строке 102. Она создает буферный объект (строка 103), присваивает указанной переменной-атрибуту (строка 109) и разрешает присваивание (строка 110), но она не записывает в буфер фактические данные. Сохранением буферных объектов в переменной `model` (строка 64) их подготовка завершается. Следующий шаг – чтение содержимого файла OBJ в буферы. Эта операция выполняется в строке 76, вызовом функции `readOBJFile()`. В первом аргументе этой функции передается адрес файла (URL), во втором – `gl`, и в третьем – объект типа `Object (model)`, содержащий буферные объекты. Эта функция решает задачи, подобные тем, что решались с помощью объекта `Image` при загрузке изображения текстуры. Они перечислены ниже:

- (2.1) Создает объект `XMLHttpRequest` (строка 117).
- (2.2) Регистрирует обработчик события для вызова по окончании загрузки файла (строка 119).
- (2.3) Создает запрос на получение файла вызовом метода `open()` (строка 124).
- (2.4) Посылает запрос (строка 125).

Объект `XMLHttpRequest`, который будет посылать HTTP-запрос веб-серверу, создается в строке 117. Регистрация обработчика события для вызова по окончании загрузки файла выполняется в строке 119. Запрос на получение файла создается в строке 124, вызовом метода `open()`. Так как целью запроса является получение файла, в первом аргументе методу `open()` передается тип запроса `'GET'`, а во втором аргументе – адрес URL файла. Последний аргумент определяет, будет ли запрос выполняться асинхронно. Наконец, в строке 125, вызовом метода `send()` запрос посылается веб-серверу.

Примечание: если вы собираетесь запускать примеры программ, использующие внешние файлы, с локального диска в браузере Chrome, добавьте параметр настройки `--allow-file-access-from-files`. Это необходимо из-за ограничений системы безопасности. По умолчанию Chrome запрещает доступ к локальным файлам, таким как `../resources/cube.obj`. В браузере Firefox имеется эквивалентный параметр `security.fileuri.strict_origin_policy`, который следует установить в значение `false` через `account:config`. Не забудьте вернуть эти настройки обратно, чтобы не оставить брешь в системе безопасности.

Как только браузер загрузит файл, он вызовет обработчик события в строке 119. В строке 120 обработчик проверяет наличие каких-либо ошибок, возникших при выполнении запроса. Значение 4 в свойстве `readyState` сообщает, что процедура загрузки закончилась успешно. Однако, если свойство `readyState` имеет любое другое значение и свойство `status` хранит код состояния 404, это означает, что указанный файл отсутствует. Код 404 – это ошибка «404 Not Found», которая отображается в окне браузера при попытке открыть несуществующую веб-страницу. Если загрузка файла увенчалась успехом, вызывается функция `onReadOBJFile()`, определение которой начинается в строке 132. Она принимает пять аргументов. Первый аргумент, `responseText`, хранит содержимое файла в виде строки. В строке 133 функция создает объект `OBJDoc`, который затем, в строке 134, используется для извлечения результатов в форме, пригодной для передачи системе WebGL. Подробнее об этом рассказывается ниже. В строке 140 объект `objDoc`, с результатами анализа содержимого файла, присваивается переменной `g_objDoc` для последующего отображения.

Объект, определяемый пользователем

Прежде чем продолжить описание программы `OBJViewer.js`, нужно разобраться с созданием собственных (определяемых пользователем) объектов в языке JavaScript. Программа `OBJViewer.js` использует такой объект для анализа содержимого файла OBJ. Пользовательские объекты в JavaScript интерпретируются точно так же, как встроенные объекты, такие как `Array` и `Date`.

Ниже приводится определение объекта `StringParser` из программы `OBJViewer.js`. Особое внимание обратите на то, как определяется **конструктор**, создающий объект, и как в этот объект добавляются методы. Конструктор – это особый метод, который вызывается при попытке создать объект с помощью оператора `new`. Ниже приводится определение конструктора объекта `StringParser`:

```
595 // Конструктор
596 var StringParser = function(str) {
597   this.str; // Сохранить строку, переданную в аргументе
598   this.index; // Позиция в строке во время анализа
599   this.init(str);
600 }
```

Конструктор можно определить как анонимную функцию (см. главу 2). Он должен иметь параметры, которые будут передаваться при создании объекта с помощью оператора `new`. В строках 597 и 598 объявляются свойства объекта, подобные свойствам встроенных объектов, таким как свойство `length` объекта `Array`. Объявление свойства выполняется с помощью ключевого слова `this`, за которым следуют точка (.) и имя свойства. В строке 599 вызывается метод инициализации `init()` этого пользовательского объекта.

Рассмотрим этот метод. Добавить метод в объект можно, указав его имя после ключевого слова `prototype`. Тело метода так же определяется как анонимная функция:

```
601 // Инициализирует объект StringParser
602 StringParser.prototype.init = function(str) {
603     this.str = str;
604     this.index = 0;
605 }
```

К свойствам, объявленным в конструкторе, можно обращаться из методов, что очень удобно. Имя `this.str` в строке 603 ссылается на свойство `this.str`, объявленное в строке 597, внутри конструктора. Имя `this.index` в строке 604 ссылается на свойство `this.index`, объявленное в строке 598. Давайте попробуем задействовать этот объект `StringParse`:

```
var sp = new StringParser('Tomorrow is another day. ');
alert(sp.str);           // Выведет: "Tomorrow is another day."
sp.str = 'Quo Vadis';    // Записшет в свойство str строку "Quo Vadis".
alert(sp.str);           // Выведет: "Quo Vadis"
sp.init('Cinderella, tonight');
alert(sp.str);           // Выведет: "Cinderella, tonight?"
```

Рассмотрим другой метод, `skipDelimiters()`, который пропускает символы-разделители (символ табуляции, пробел, круглые скобки и кавычки "):

```
608 StringParser.prototype.skipDelimiters = function() {
609     for(var i = this.index, len = this.str.length; i < len; i++) {
610         var c = this.str.charAt(i);
611         // Пропустить символы-разделители
612         if (c == '\t' || c == ' ' || c == '(' || c == ')' || c == '"') continue;
613         break;
614     }
615     this.index = i;
616 }
```

Метод `charAt()`, что вызывается в строке 610, поддерживается объектами `String`, представляющими строки, и возвращает символ в позиции, которая определяется аргументом.

Теперь посмотрим, как реализован анализ содержимого файла OBJ в программе `OBJViewer.js`.

Пример программы (реализация анализа содержимого файла)

Программа `OBJViewer.js` осуществляет построчный анализ содержимого файла OBJ и преобразует его в структуру, изображенную на рис. 10.26. Каждый прямоугольник на рис. 10.26 – это пользовательский объект. Несмотря на то, что реализация анализа содержимого файла в программе `OBJViewer.js` выглядит достаточно сложной, сама процедура анализа достаточно проста. Сложность обусловлена лишь наличием повторяющихся строк кода. Давайте рассмотрим, как выполняется анализ в принципе, так как это поможет вам понять весь процесс в целом.

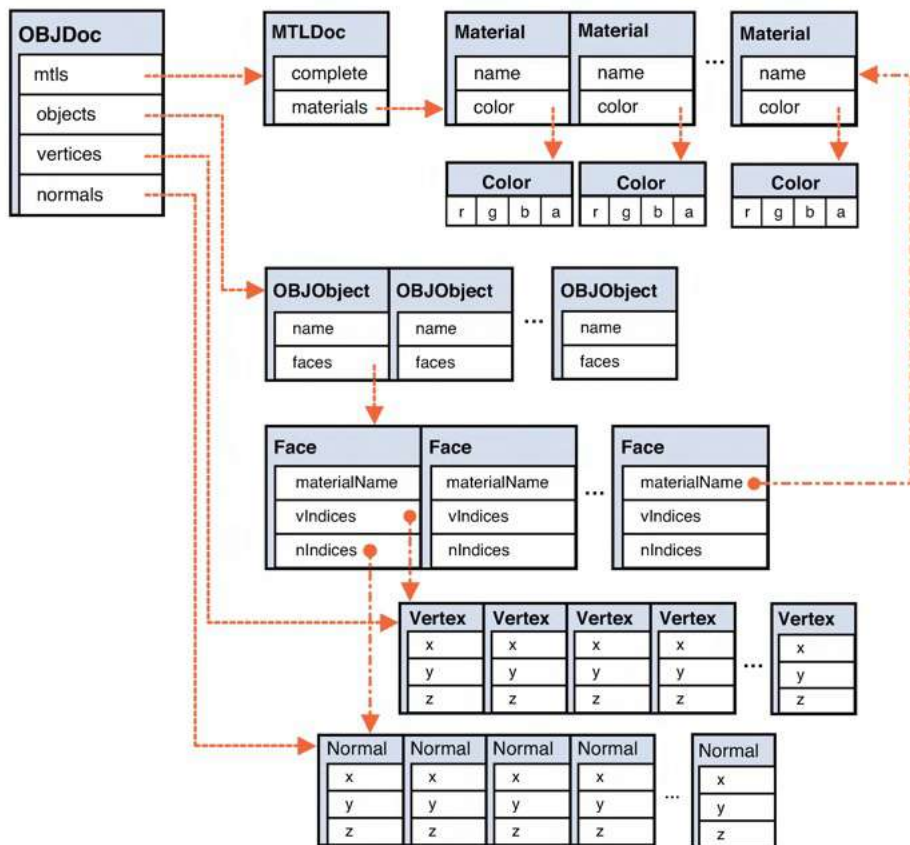


Рис. 10.26. Структура данных, получаемая в результате анализа файла OBJ

В листинге 10.21 представлен основной код, реализующий анализ файла в программе `OBJViewer.js`.

Листинг 10.21. `OBJViewer.js` (анализ файла)

```

214 // OBJDoc object
215 // Конструктор
216 var OBJDoc = function(fileName) {
217   this.fileName = fileName;
218   this.mtls = new Array(0); // Инициализировать свойство для материалов
219   this.objects = new Array(0); // Инициализировать свойство для объектов
220   this.vertices = new Array(0); // Инициализировать свойство для вершин
221   this.normals = new Array(0); // Инициализировать свойство для нормалей
222 }
223
224 // Анализирует файл OBJ
225 OBJDoc.prototype.parse = function(fileString, scale, reverseNormal) {
226   var lines = fileString.split('\n'); // Разбить на строки

```



```
227 lines.push(null); // Добавить в конец null
228 var index = 0; // Инициализировать индекс строки
229
230 var currentObject = null;
231 var currentMaterialName = "";
232
233 // Выполнить построчный анализ
234 var line; // Содержимое анализируемой строки
235 var sp = new StringParser(); // Создать StringParser
236 while ((line = lines[index++]) != null) {
237     sp.init(line); // Инициализировать StringParser
238     var command = sp.getWord(); // Получить команду
239     if(command == null) continue; // проверить на равенство null
240
241     switch(command){
242     case '#':
243         continue; // Пропустить комментарий
244     case 'mtllib': // Прочитать описание материала
245         var path = this.parseMtlLib(sp, this.fileName);
246         var mtl = new MTLDoc(); // Создать экземпляр MTL
247         this.mtls.push(mtl);
248         var request = new XMLHttpRequest();
249         request.onreadystatechange = function() {
250             if (request.readyState == 4) {
251                 if (request.status != 404) {
252                     onReadMTLFile(request.responseText, mtl);
253                 }else{
254                     mtl.complete = true;
255                 }
256             }
257         }
258         request.open('GET', path, true); // Создать запрос на получение файла
259         request.send(); // Послать запрос
260         continue; // Перейти к следующей строке
261     case 'o':
262     case 'g': // Прочитать имя объекта Object
263         var object = this.parseObjectName(sp);
264         this.objects.push(object);
265         currentObject = object;
266         continue; // Перейти к следующей строке
267     case 'v': // Прочитать координаты вершины
268         var vertex = this.parseVertex(sp, scale);
269         this.vertices.push(vertex);
270         continue; // Перейти к следующей строке
271     case 'vn': // Прочитать нормаль
272         var normal = this.parseNormal(sp);
273         this.normals.push(normal);
274         continue; // Перейти к следующей строке
275     case 'usemtl': // Прочитать название материала
276         currentMaterialName = this.parseUsemtl(sp);
277         continue; // Перейти к следующей строке
278     case 'f': // Прочитать определение грани
279         var face = this.parseFace(sp, currentMaterialName, this.vertices,
280                                     reverse);
281         currentObject.addFace(face);
```

```

281     continue; // Перейти к следующей строке
282 }
283 }
284
285 return true;
286 }

```

В строках с 216 по 222 определяется конструктор объекта `OBJDoc`, имеющего пять свойств, которые будут использоваться для сохранения результатов анализа. Фактический анализ выполняется методом `parse()`, определение которого начинается в строке 225. Содержимое файла `OBJ` передается методу в аргументе `fileString` в виде единой строки. Метод `parse()` разбивает ее на отдельные фрагменты вызовом строкового метода `split()`. Этот метод разбивает строку по символам, указанным в аргументе. Как можно видеть в строке 226, аргумент определяет символ «\n» (перевод строки), соответственно в свойстве `this.line` все фрагменты сохраняются в виде массива. В строке 227 в конец массива добавляется пустая ссылка (`null`), чтобы потом проще было определить конец массива. Свойство `this.index` хранит число проанализированных строк и в строке 228 инициализируется значением 0.

В предыдущем разделе вы уже видели реализацию объекта `StringParser`, который создается в строке 235. Этот объект используется для анализа содержимого одного фрагмента.

Теперь все готово к анализу файла `OBJ`. В строке 236 все фрагменты поочередно сохраняются в переменной `line` обращением к массиву `this.lines[this.index++]`. В строке 237 выполняется запись фрагмента в `sp` (объект типа `StringParser`). В строке 238 из фрагмента извлекается первое слово, вызовом `sp.getWord()`, и сохраняется в переменной `command`. Используемые нами методы перечислены в табл. 10.3, где под словом «слово» подразумевается часть строки, окруженная разделителями (табуляция, пробел, (,) или ").

Таблица 10.3. Методы, поддерживаемые объектом `StringParser`

Метод	Описание
<code>StringParser.init(str)</code>	Инициализирует <code>StringParser</code> , готовя его к анализу <code>str</code> .
<code>StringParser.getWord()</code>	Возвращает слово.
<code>StringParser.skipToNextWord()</code>	Выполняет переход к следующему слову.
<code>StringParser.getInt()</code>	Извлекает слово и преобразует его в целое число.
<code>StringParser.getFloat()</code>	Извлекает слово и преобразует его в вещественное число.

Инструкция `switch` в строке 241 проверяет переменную с командой `command`, чтобы определить, как обрабатывать последующие строки в файле `OBJ`.

Если команда является символом `#` (строка 242), вся строка интерпретируется как комментарий. Поэтому в строке 243 выполняется инструкция `continue`, чтобы пропустить эту строку.

В случае команды `mtllib` (строка 241) строка интерпретируется как ссылка на файл MTL. В строке 245 генерируется путь к файлу. В строке 246 создается объект типа `MTLDoc` для хранения информации о материале из файла MTL, и в строке 247 этот объект сохраняется в массиве `this.mtls`. Затем, в строках с 248 по 259 выполняется чтение файла, тем же способом, каким выполнялось чтение файла OBJ. Анализ файла MTL выполняется обработчиком `onReadMTLfile()`, который вызывается по окончании загрузки.

В случае команды `o` (строка 261) или `g` (строка 262) строка интерпретируется как именованный объект или группа. В строке 263 выполняется анализ оставшейся части фрагмента и полученный результат в виде объекта типа `OBJObject` сохраняется в массиве `this.objects` (строка 264) и в переменной `currentObject` (строка 265).

В случае команды `v` строка интерпретируется, как координаты вершины. Строка 268 выполняет анализ (x, y, z) и возвращает результат в виде объекта типа `Vertex`. В строке 269 этот объект сохраняется в массиве `this.vertices`.

Команда `f` говорит о том, что строка содержит определение грани. Строка 279 анализирует это определение и возвращает результат в виде объекта типа `Face`. Данный объект сохраняется в переменной `currentObject`.

А теперь посмотрим, как выполняется анализ координат вершины в функции `parseVertex()`, определение которой приводится в листинге 10.22.

Листинг 10.22. `OBJViewer.js (parseVertex())`

```
302 OBJDoc.prototype.parseVertex = function(sp, scale) {
303     var x = sp.getFloat() * scale;
304     var y = sp.getFloat() * scale;
305     var z = sp.getFloat() * scale;
306     return (new Vertex(x, y, z));
307 }
```

Строка 303 извлекает значение X , вызывая `sp.getFloat()`. Значение масштабируется умножением на `scale`, если модель слишком маленькая или слишком большая. После извлечения всех трех координат, в строке 306 создается объект `Vertex` и возвращается вызывающей программе.

После завершения анализа файлов OBJ и MTL массивы с координатами вершин, значениями цвета, нормальными и индексами образуют структуру, изображенную на рис. 10.26. Затем вызывается обработчик `onReadComplete()`, который записывает полученные данные в буферные объекты (см. листинг 10.23).

Листинг 10.23. `OBJViewer.js (onReadComplete())`

```
176 // Файл OBJ прочитан полностью
177 function onReadComplete(gl, model, objDoc) {
178     // Получить координаты вершин и цвета из файла OBJ
179     var drawingInfo = objDoc.getDrawingInfo();
180
181     // Записать данные в буферные объекты
```



```

182 gl.bindBuffer(gl.ARRAY_BUFFER, model.vertexBuffer);
183 gl.bufferData(gl.ARRAY_BUFFER, drawingInfo.vertices, gl.STATIC_DRAW);
184
185 gl.bindBuffer(gl.ARRAY_BUFFER, model.normalBuffer);
186 gl.bufferData(gl.ARRAY_BUFFER, drawingInfo.normals, gl.STATIC_DRAW);
187
188 gl.bindBuffer(gl.ARRAY_BUFFER, model.colorBuffer);
189 gl.bufferData(gl.ARRAY_BUFFER, drawingInfo.colors, gl.STATIC_DRAW);
190
191 // Записать индексы в буферный объект
192 gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, model.indexBuffer);
193 gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, drawingInfo.indices, gl.STATIC_DRAW);
194
195 return drawingInfo;
196 }

```

Этот метод достаточно прост. В строке 178 он извлекает из объекта `objDoc` информацию, необходимую для рисования и полученную в результате анализа файла OBJ. Строки 183, 186, 189 и 193 записывают вершины, нормали, цвета и индексы в соответствующие буферные объекты.

Функция `getDrawingInfo()`, определение которой начинается в строке 451 (см. листинг 10.24), извлекает вершины, нормали, цвета и индексы из объекта `objDoc`.

Листинг 10.24. OBJViewer.js (извлечение информации, необходимой для рисования)

```

450 // Извлекает информацию для рисования трехмерной модели
451 OBJDoc.prototype.getDrawingInfo = function() {
452     // Создать массив координат вершин, нормалей, цветов и индексов
453     var numIndices = 0;
454     for (var i = 0; i < this.objects.length; i++){
455         numIndices += this.objects[i].numIndices;
456     }
457     var numVertices = numIndices;
458     var vertices = new Float32Array(numVertices * 3);
459     var normals = new Float32Array(numVertices * 3);
460     var colors = new Float32Array(numVertices * 4);
461     var indices = new Uint16Array(numIndices);
462
463     // Определить вершины, нормали и цвета
464     var index_indices = 0;
465     for (var i = 0; i < this.objects.length; i++){
466         var object = this.objects[i];
467         for (var j = 0; j < object.faces.length; j++){
468             var face = object.face[j];
469             var color = this.findColor(face.materialName);
470             var faceNormal = face.normal;
471             for (var k = 0; k < face.vIndices.length; k++){
472                 // Установить индекс
473                 indices[index_indices] = index_indices;
474                 // Скопировать вершину
475                 var vIdx = face.vIndices[k];
476                 var vertex = this.vertices[vIdx];
477                 vertices[index_indices * 3 + 0] = vertex.x;

```



```
478     vertices[index_indices * 3 + 1] = vertex.y;
479     vertices[index_indices * 3 + 2] = vertex.z;
480     // Скопировать цвет
481     colors[index_indices * 4 + 0] = color.r;
482     colors[index_indices * 4 + 1] = color.g;
483     colors[index_indices * 4 + 2] = color.b;
484     colors[index_indices * 4 + 3] = color.a;
485     // Скопировать нормаль
486     var nIdx = face.nIndices[k];
487     if(nIdx >= 0){
488         var normal = this.normals[nIdx];
489         normals[index_indices * 3 + 0] = normal.x;
490         normals[index_indices * 3 + 1] = normal.y;
491         normals[index_indices * 3 + 2] = normal.z;
492     }else{
493         normals[index_indices * 3 + 0] = faceNormal.x;
494         normals[index_indices * 3 + 1] = faceNormal.y;
495         normals[index_indices * 3 + 2] = faceNormal.z;
496     }
497     index_indices ++;
498 }
499 }
500 }
501
502 return new DrawingInfo(vertices, normals, colors, indices);
503 };
```

В строке 454, с помощью цикла `for`, вычисляется число индексов. Затем в строках с 458 по 461 создаются типизированные массивы для хранения вершин, нормалей, цветов и индексов, которые присваиваются соответствующим буферным объектам. Размер каждого массива определяется числом индексов, вычисленным в строке 454.

Программа выполняет обход объектов `OBJObject` и хранящихся в них объектов `Face` в порядке, как показано на рис. 10.28, и сохраняет информацию в массивах `vertices`, `colors` и `indices`.

Инструкция цикла `for` в строке 465 извлекает каждый объект `OBJObject` из результатов, полученных ранее в ходе анализа. Инструкция цикла `for` в строке 467 делает то же самое с объектами `Face`, хранящимися в объектах `OBJObject`, и с каждым объектом `Face` выполняет следующие действия:

1. В строке 469 выполняется поиск цвета грани по свойству `materialName` и сохраняет его в переменной `color`. Строка 470 сохраняет нормаль грани в переменной `faceNormal`.
2. Инструкция цикла `for` в строке 471 извлекает индекс вершины из описания грани, сохраняет координаты вершины в массиве `vertices` (строки с 477 по 479), а также сохраняет компоненты цвета RGB в массиве `colors` (строки с 482 по 484). Код, начиная со строки 486, обрабатывает нормали. Файлы `OBJ` могут и не содержать описания нормалей, поэтому в строке 487 проверяется их наличие. Если информация о нормалях присутствует в файле `OBJ`, строки с 487 по 489 сохраняют ее в массиве `normals`. В про-

тивном случае выполняются строки с 492 по 494, генерирующие нормали.

После выполнения этих действий со всеми объектами `OBJObject`, программа готова приступить к рисованию. Строка 502 возвращает информацию для рисования модели в виде объекта `DrawingInfo`, содержащего буферные объекты, описанные выше.

Несмотря на сжатое объяснение, на данном этапе вы должны достаточно полно понимать, как прочитать содержимое файла OBJ, проанализировать и отобразить с помощью WebGL. Если потребуется прочитать несколько файлов, описывающих единственную сцену, вам достаточно будет повторить эту процедуру. В каталоге `resource` вы найдете еще несколько файлов OBJ, с которыми вы сможете поэкспериментировать, чтобы убедиться, что все поняли правильно (см. рис. 10.27).



Рис. 10.27. Различные трехмерные модели

Обработка ситуации потери контекста

Для отображения графики, система WebGL использует видеокарту компьютера, которая является разделяемым ресурсом, управляемым операционной системой. Иногда в процессе работы могут возникать ситуации, приводящие к «потере» ресурса, в результате которых теряется информация, хранящаяся в видеокарте. Такие ситуации могут возникать, например, когда другая программа пытается выполнять операции с видеокартой или когда компьютер переходит в режим ожидания. В таких ситуациях информация, которую система WebGL использует для рисования, ее «контекст», может быть потеряна. Хорошим примером может служить ситуация, когда WebGL-программа выполняется на ноутбуке или смартфоне, и устройство переходит в режим ожидания. Часто перед переходом в этот режим выводится сообщение об ошибке. Когда устройство возвращается в нормальный режим функционирования, система возвращается в исходное состояние, но браузер, под управлением которого выполняется WebGL-программа, может ничего не отображать на экране, как показано на рис. 10.28 справа. Так как веб-страница в этом примере программы имеет белый фон, веб-браузер отображает чистое, белое окно.

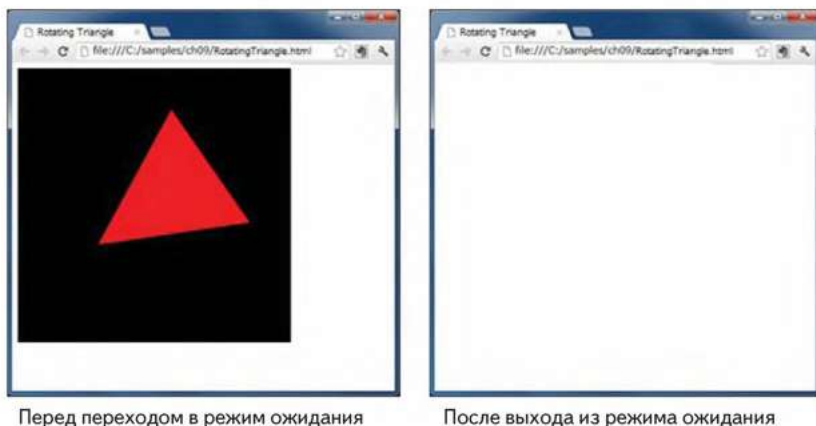


Рис. 10.30. WebGL-программа прекращает рисовать после возврата компьютера из режима ожидания

Например, если запустить программу `RotatingTriangle`, в консоли может появиться следующее сообщение:

```
WebGL error CONTEXT_LOST_WEBGL in uniformMatrix4fv({object WebGLUniformLocation, false, [object Float32Array]}
```

Это сообщение указывает, что во время выполнения программы, в методе `gl.uniformMatrix4fv()` возникла ошибка, либо перед входом в режим ожидания, либо по возвращении из него. Содержимое сообщения об ошибке могут отличаться, в зависимости от того, что программа пыталась сделать в во время режима ожидания. В этом разделе мы расскажем, как решить эту проблему.

Как реализовать обслуживание ситуации потери контекста

Как уже говорилось выше, программа может потерять контекст по самым разным причинам. Однако, WebGL поддерживает два события, информирующих об изменении состояния системы: **событие потери контекста** (`webglcontextlost`) и **событие восстановления контекста** (`webglcontextrestored`). См. табл. 10.4.

Таблица 10.4. События потери и восстановления контекста

Событие	Описание
<code>webglcontextlost</code>	Возникает, когда WebGL теряет контекст отображения
<code>webglcontextrestored</code>	Возникает, когда браузер повторно инициализирует систему WebGL

Когда возникает событие потери контекста, контекст отображения (то есть, ссылка `gl` в примерах программ), полученная вызовом `getWebGLContext()`, становится недействительным, и любая операция с контекстом `gl` вызывает ошибку.

К таким операциям относятся: создание буферных объектов и объектов текстуры, инициализация шейдеров, установка цвета очистки и многие другие. После того, как браузер повторно инициализирует систему WebGL, возбуждается событие восстановления контекста, в ответ на которое программа должна повторно выполнить все эти операции. Остальные переменные в программе на JavaScript не затрагиваются и могут использоваться как обычно.

Прежде чем перейти к примеру программы, нам нужно познакомиться с методом `addEventListener()` элемента `<canvas>`, с помощью которого можно зарегистрировать обработчик события потери и восстановления контекста. Такая необходимость объясняется тем, что элемент `<canvas>` не поддерживает специализированные свойства, которые позволяли бы регистрировать обработчики привычным способом. Вспомните, как в предыдущих примерах мы использовали свойство `onmousedown` элемента `<canvas>` для регистрации обработчика событий от мыши.

`canvas.addEventListener(type, handler, useCapture)`

Регистрирует обработчик события `handler` в элементе `<canvas>`.

Параметры	<code>type</code>	Определяет имя обрабатываемого события (строка).
	<code>handler</code>	Определяет обработчик события. Эта функция будет вызываться с единственным аргументом (объект <code>event</code>).
	<code>useCapture</code>	Определяет необходимость захвата события (логическое значение). Если имеет значение <code>true</code> , событие не будет доставляться другим элементам. Если имеет значение <code>true</code> , событие будет передаваться другим элементам

Возвращаемое значение нет

Пример программы (*RotatingTriangle_contextLost.js*)

В этом разделе мы напишем программу `RotatingTriangle_contextLost`, которая является версией программы `RotatingTriangle`, поддерживающей возможность восстановления контекста после его потери (скриншот показан на рис. 10.28). Исходный код программы приводится в листинге 10.25.

Листинг 10.25. `RotatingTriangle_contextLost.js`

```
1 // RotatingTriangle_contextLost.js
  ...
16 function main() {
17   // Получить ссылку на элемент <canvas>
18   var canvas = document.getElementById('webgl');
```



```

19
20 // Зарегистрировать обработчик событий потери и восстановления контекста
21 canvas.addEventListener('webglcontextlost', contextLost, false);
22 canvas.addEventListener('webglcontextrestored', function(ev)
    ➤{ start(canvas); }, false);
23
24 start(canvas); // Выполнить процедуры, связанные с WebGL
25 }
...
29 // Текущий угол поворота
30 var g_currentAngle = 0.0; // Прежде локальная переменная, теперь глобальная
31 var g_requestID; // Возвращаемое значение requestAnimationFrame()
32
33 function start(canvas) {
34 // Получить контекст отображения WebGL
35 var gl = getWebGLContext(canvas);
36
37 ...
41 // Инициализировать шейдеры
42 if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
43     ...
44 }
45
46
47 var n = initVertexBuffers(gl); // Определить координаты вершин
48
49 ...
55 // Получить ссылку на u_ModelMatrix
56 var u_ModelMatrix = gl.getUniformLocation(gl.program, 'u_ModelMatrix');
57
58 ...
62 var modelMatrix = new Matrix4(); // Создать матрицу модели
63
64 var tick = function() { // Начать рисование
65     g_currentAngle = animate(g_currentAngle); // Изменить угол поворота
66     draw(gl, n, g_currentAngle, modelMatrix, u_ModelMatrix);
67     g_requestID = requestAnimationFrame(tick, canvas);
68 };
69 tick();
70 }
71
72 function contextLost(ev) { // Обработчик события потери контекста
73     cancelAnimationFrame(g_requestID); // Остановить анимацию
74     ev.preventDefault(); // Предотвратить обработку по умолчанию
75 }

```

Обработка события потери контекста никак не связана с шейдерами, поэтому сосредоточимся на JavaScript-функции `main()`, определение которой начинается в строке 16. В строке 21 выполняется регистрация обработчика события потери контекста, а в строке 22 регистрируется обработчик события восстановления контекста. Функция `main()` завершается вызовом `start()` в строке 24.

Определение функции `start()` начинается в строке 33. Она выполняет те же шаги, что и функция `main()` в программе `RotatingTriangle.js`. Их понадобится выполнить повторно после события потери контекста, для обработки которого в `RotatingTriangle_contextLost.js` внесено два изменения, по сравнению с оригинальной версией `RotatingTriangle.js`.

Во-первых, текущий угол поворота (используется в строке 65) теперь хранится не в локальной, а в глобальной переменной `g_currentAngle` (объявляется в строке 30). Это позволяет возобновить вращение треугольника, начиная с текущего угла, при появлении события восстановления контекста. Во-вторых, в строке 67 в глобальной переменной `g_requestID` (объявляется в строке 31) сохраняется возвращаемое значение `requestAnimationFrame()`. Эта переменная используется для отмены регистрации функции при потере контекста.

Рассмотрим фактические обработчики событий. Обработчик события потери контекста, `contextLost()`, определяется, начиная со строки 72, и состоит всего из двух строк кода. Строка 73 отменяет регистрацию функции, используемой для обработки анимации, что гарантирует предотвращение дальнейших попыток обратиться к контексту, пока он не будет восстановлен. Строка 74 предотвращает выполнение браузером действий по обработке этого события, предусмотренные по умолчанию. Это совершенно необходимо потому, что по умолчанию, в ответ на событие потери контекста, браузер запрещает возбуждение события восстановления контекста. Однако нам нужно это событие, поэтому мы должны запретить действия по умолчанию.

Обработчик события восстановления контекста (строка 22) выглядит еще проще: он просто вызывает функцию `start()`, которая заново приобретет контекст WebGL.

Обратите внимание, что когда возникает событие потери контекста, в консоль всегда выводится следующее сообщение:

```
WARNING: WebGL content on the page might have caused the graphics card to reset  
(ВНИМАНИЕ: произошел сброс графической карты, возможно, вызванный WebGL-содержимым)
```

Реализуя подобные обработчики, в своих WebGL-приложениях вы сможете обрабатывать ситуации потери контекста WebGL.

В заключение

В этой главе мы познакомились со множеством разнообразных приемов, которые безусловно пригодятся вам при создании собственных WebGL-приложений. Из-за ограниченного пространства в книге, мы постарались максимально сократить наши пояснения, но при этом дать достаточно информации, чтобы вы могли самостоятельно использовать эти приемы. Несмотря на то, что существует еще великое множество приемов, которые могли бы оказаться вам полезными, мы выбрали эти, потому что они помогут вам начать применять полученные в этой книге знания для создания собственных приложений с трехмерной графикой.

Как вы имели возможность убедиться, WebGL – это мощный инструмент для создания приложений с трехмерной графикой, позволяющий создавать ошеломительные по своей сложности трехмерные изображения. Наша цель в этой книге состояла в том, чтобы дать вам пошаговое введение в основы WebGL и заложить прочный фундамент знаний, опираясь на который вы сможете начать создавать

собственные WebGL-приложения и продолжить дальнейшие исследования. Существует множество источников информации, которые вы могли бы использовать. Однако мы надеемся, что в своих исследованиях мира WebGL вы еще вернетесь к этой книге и оцените ее, уже как справочное руководство.



ПРИЛОЖЕНИЕ А.

В WebGL нет необходимости использовать рабочий и фоновый буферы

Те из вас, кто имеет опыт разработки приложений с использованием библиотеки OpenGL, могли заметить, что ни в одном из примеров, приводившихся в этой книге, не использовался прием перестановки буферов цвета, который является обязательным при работе с большинством реализаций OpenGL.

Как вы наверняка знаете, OpenGL использует два буфера цвета: «рабочий» и «фоновый», при этом рабочим называют буфер, содержимое которого отображается на экране. Обычно, когда приложение использует OpenGL, оно рисует графику в фоновом буфере цвета. Когда ее необходимо вывести на экран, требуется скопировать содержимое фонового буфера в рабочий. Если рисовать непосредственно в рабочем буфере, на экране будут появляться визуальные артефакты (например, мерцание, полосы и другие) из-за того, что обновление изображения на экране будет происходить до того как данные в буфере будут готовы к отображению.

Для поддержки двойной буферизации в OpenGL имеется механизм перестановки фонового и рабочего буферов. В некоторых системах этот механизм действует автоматически; в других, после создания изображения в фоновом буфере, эту операцию требуется выполнять явно, вызывая такие функции, как `glutSwapBuffers()` или `eglSwapBuffers()`. Например, типичные приложения с поддержкой OpenGL часто имеют пользовательскую функцию «display», как показано ниже:

```
void display(void) {  
    // Очистить буферы цвета и глубины  
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
    draw(); // Нарисовать сцену  
    glutSwapBuffers(); // Выполнить перестановку буферов  
}
```

Система WebGL, напротив, опирается на механизм обновления изображения, реализованный в браузере, который действует автоматически, избавляя вас от необходимости делать это вручную. Взгляните на рис. А.1 (это рис. 2.10 из главы 2):

когда WebGL-приложение рисует что-либо в буфере цвета, браузер определяет этот факт и выводит содержимое на экран. По этой причине WebGL поддерживает только один буфер цвета.

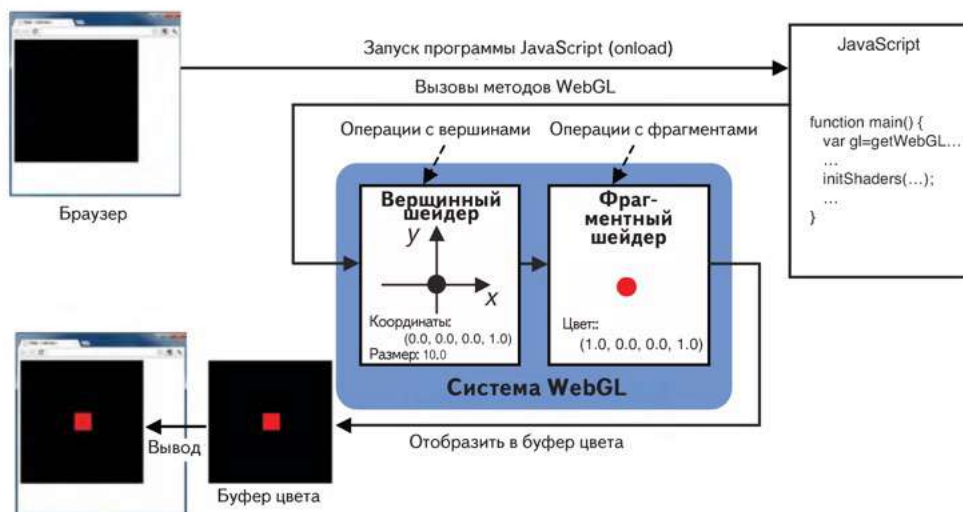


Рис. А.1. Последовательность операций, от запуска программы JavaScript до отображения результатов в окне браузера

Этот подход работает, благодаря тому, что все примеры WebGL-приложений в этой книге запускаются под управлением браузера вызовом метода на JavaScript.

Так как программы не являются автономными, у браузера есть возможность определить факт изменения содержимого буфера цвета программой на JavaScript. Если содержимое изменилось, браузер автоматически выведет его на экран.

Например, в программе `HelloPoint1` мы вызывали JavaScript-функцию (`main()`) из документа HTML (`HelloPoint1.html`), как показано ниже:

```
<body onload="main()">
```

Эта строка заставляет браузер выполнить функцию `main()` после загрузки элемента `<body>`. Внутри функции `main()` выполняется операция рисования, которая изменяет содержимое буфера цвета.

```
main(){
  ...
  // Нарисовать точку
  gl.drawArrays(gl.POINTS, 0, 1);
}
```

Когда функция `main()` завершится, управление вновь вернется браузеру, вызвавшему ее. Затем браузер проверит содержимое буфера цвета, и если обнаружит изменения, выведет его на экран. Такой принцип действия имеет один положительный побочный эффект: браузер автоматически объединяет буфер цвета с

остальной веб-страницей, давая нам возможность встраивать трехмерную графику. Обратите внимание, что программа `HelloPoint1` выводит единственный элемент `<canvas>` на страницу, потому что в файле `HelloPoint1.html` нет никаких других элементов, кроме элемента `<canvas>`.

Из вышесказанного следует, что вызывая методы, возвращающие управление браузеру, такие как `alert()` или `confirm()`, мы тем самым способствуем выводу содержимого буфера цвета на экран. Возможно, это не всегда будет соответствовать вашим намерениями, поэтому будьте внимательны при использовании таких методов в WebGL-программах.

Браузер действует точно так же, когда программа на JavaScript рисует что-то в обработчиках событий. Это объясняется тем, что обработчики вызываются браузером и по окончании работы возвращают управление браузеру.



ПРИЛОЖЕНИЕ В.

Встроенные функции в языке GLSL ES 1.0

В этом приложении описываются все встроенные функции, поддерживаемые в языке GLSL ES 1.0, включая множество функций, не упоминавшихся в этой книге, но которые часто используются при программировании шейдеров.

Обратите внимание, что все функции, кроме функций для работы с текстурами, выполняют операции с векторами или матрицами поэлементно. Например,

```
vec2 deg = vec2(60, 80);  
vec2 rad = radians(deg);
```

В этих примерах компонентам массива `rad` будут присвоены значения, соответствующие величинам углов в 60 и 80 градусов, выраженные в радианах.

Функции для работы с угловыми величинами и тригонометрические функции

Синтаксис	Описание
<code>float radians(float degree)</code> <code>vec2 radians(vec2 degree)</code> <code>vec3 radians(vec3 degree)</code> <code>vec4 radians(vec4 degree)</code>	Преобразует градусы в радианы; то есть, $\pi * \text{degree} / 180$.
<code>float degrees(float radian)</code> <code>vec2 degrees(vec2 radian)</code> <code>vec3 degrees(vec3 radian)</code> <code>vec4 degrees(vec4 radian)</code>	Преобразует радианы в градусы; то есть, $180 * \text{radian} / \pi$.
<code>float sin(float angle)</code> <code>vec2 sin(vec2 angle)</code> <code>vec3 sin(vec3 angle)</code> <code>vec4 sin(vec4 angle)</code>	Стандартная тригонометрическая функция вычисления синуса. Параметр <code>angle</code> выражен в радианах. Возвращаемое значение находится в диапазоне $[-1, 1]$.

Синтаксис	Описание
float cos(float angle) vec2 cos(vec2 angle) vec3 cos(vec3 angle) vec4 cos(vec4 angle)	Стандартная тригонометрическая функция вычисления косинуса. Параметр <code>angle</code> выражен в радианах. Возвращаемое значение находится в диапазоне $[-1, 1]$.
float tan(float angle) vec2 tan(vec2 angle) vec3 tan(vec3 angle) vec4 tan(vec4 angle)	Стандартная тригонометрическая функция вычисления тангенса. Параметр <code>angle</code> выражен в радианах.
float asin(float x) vec2 asin(vec2 x) vec3 asin(vec3 x) vec4 asin(vec4 x)	<i>Арксинус</i> . Возвращает угол (в радианах), синус которого равен <code>x</code> . Возвращаемое значение находится в диапазоне $[-\pi/2, \pi/2]$. Для $x > -1$ или $x > +1$ результат неопределен.
float acos(float x) vec2 acos(vec2 x) vec3 acos(vec3 x) vec4 acos(vec4 x)	<i>Аркосинус</i> . Возвращает угол (в радианах), косинус которого равен <code>x</code> . Возвращаемое значение находится в диапазоне $[0, \pi]$. Для $x > -1$ или $x > +1$ результат неопределен.
float atan(float y, float x) vec2 atan(vec2 y, vec2 x) vec3 atan(vec3 y, vec3 x) vec4 atan(vec4 y, vec4 x)	<i>Арктангенс</i> . Возвращает угол (в радианах), тангенс которого равен <code>y/x</code> . Знаки параметров <code>x</code> и <code>y</code> используются для определения квадранта угла. Возвращаемое значение находится в диапазоне $[-\pi, \pi]$. Результат неопределен, если оба параметра, <code>x</code> и <code>y</code> , имеют значение 0. <i>Обратите внимание:</i> для векторов операция выполняется покомпонентно.
float atan(float y_over_x) vec2 atan(vec2 y_over_x) vec3 atan(vec3 y_over_x) vec4 atan(vec4 y_over_x)	<i>Арктангенс</i> . Возвращает угол (в радианах), тангенс которого равен <code>y_over_x</code> . Возвращаемое значение находится в диапазоне $[-\pi/2, \pi/2]$. <i>Обратите внимание:</i> для векторов операция выполняется покомпонентно.

Экспоненциальные функции

Синтаксис	Описание
float pow(float x, float y) vec2 pow(vec2 x, vec2 y) vec3 pow(vec3 x, vec3 y) vec4 pow(vec4 x, vec4 y)	Возвращает <code>x</code> в степени <code>y</code> , то есть x^y . Результат неопределен, если $x < 0$. Результат неопределен, если $x = 0$ и $y \leq 0$. Обратите внимание: для векторов операция выполняется покомпонентно.
float exp(float x) vec2 exp(vec2 x) vec3 exp(vec3 x) vec4 exp(vec4 x)	Возвращает степень числа <code>e</code> , то есть e^x .

Синтаксис	Описание
float log(float x) vec2 log(vec2 x) vec3 log(vec3 x) vec4 log(vec4 x)	Возвращает натуральный логарифм числа x ; то есть, такое значение y , которое удовлетворяет тождеству $x = e^y$. Результат неопределен, если $x \leq 0$
float exp2(float x) vec2 exp2(vec2 x) vec3 exp2(vec3 x) vec4 exp2(vec4 x)	Возвращает 2 в степени x ; то есть, 2^x .
float log2(float x) vec2 log2(vec2 x) vec3 log2(vec3 x) vec4 log2(vec4 x)	Возвращает логарифм по основанию 2 числа x ; то есть, такое значение y , которое удовлетворяет тождеству $x = 2^y$. Результат неопределен, если $x \leq 0$
float sqrt(float x) vec2 sqrt(vec2 x) vec3 sqrt(vec3 x) vec4 sqrt(vec4 x)	Возвращает $\sqrt{x}$
float inversesqrt(float x) vec2 inversesqrt(vec2 x) vec3 inversesqrt(vec3 x) vec4 inversesqrt(vec4 x)	Возвращает $1/\sqrt{x}$

Общие функции

Синтаксис	Описание
float abs(float x) vec2 abs(vec2 x) vec3 abs(vec3 x) vec4 abs(vec4 x)	Возвращает неотрицательное значение x , независимо от его знака; то есть, возвращает x , если $x \geq 0$, иначе возвращает $-x$.
float sign(float x) vec2 sign(vec2 x) vec3 sign(vec3 x) vec4 sign(vec4 x)	Возвращает 1.0, если $x > 0$; 0.0, если $x = 0$ и -1.0, если $x < 0$.
float floor(float x) vec2 floor(vec2 x) vec3 floor(vec3 x) vec4 floor(vec4 x)	Возвращает значение, равное ближайшему целому, которое меньше или равно x .
float ceil(float x) vec2 ceil(vec2 x) vec3 ceil(vec3 x) vec4 ceil(vec4 x)	Возвращает значение, равное ближайшему целому, которое больше или равно x .
float fract(float x) vec2 fract(vec2 x) vec3 fract(vec3 x) vec4 fract(vec4 x)	Возвращает дробную часть числа x ; то есть, $x - \text{floor}(x)$.

Синтаксис	Описание
float mod(float x, float y) vec2 mod(vec2 x, vec2 y) vec3 mod(vec3 x, vec3 y) vec4 mod(vec4 x, vec4 y) vec2 mod(vec2 x, float y) vec3 mod(vec3 x, float y) vec4 mod(vec4 x, float y)	Деление по модулю. Возвращает остаток от деления x на y; то есть, $(x - y * \text{floor}(x / y))$. Для двух положительных чисел x и y, $\text{mod}(x, y)$ вернет остаток от деления x на y. <i>Обратите внимание:</i> для векторов операция выполняется покомпонентно.
float min(float x, float y) vec2 min(vec2 x, vec2 y) vec3 min(vec3 x, vec3 y) vec4 min(vec4 x, vec4 y) vec2 min(vec2 x, float y) vec3 min(vec3 x, float y) vec4 min(vec4 x, float y)	Возвращает наименьшее значение; то есть, y, если $y < x$, иначе вернет x. <i>Обратите внимание:</i> для векторов операция выполняется покомпонентно.
float max(float x, float y) vec2 max(vec2 x, vec2 y) vec3 max(vec3 x, vec3 y) vec4 max(vec4 x, vec4 y) vec2 max(vec2 x, float y) vec3 max(vec3 x, float y) vec4 max(vec4 x, float y)	Возвращает наибольшее значение; то есть, y, если $x < y$, иначе вернет x. <i>Обратите внимание:</i> для векторов операция выполняется покомпонентно.
float clamp(float x, float minVal, float maxVal) vec2 clamp(vec2 x, vec2 minVal, vec2 maxVal) vec3 clamp(vec3 x, vec3 minVal, vec3 maxVal) vec4 clamp(vec4 x, vec4 minVal, vec4 maxVal) vec2 clamp(vec2 x, float minVal, float maxVal) vec3 clamp(vec3 x, float minVal, float maxVal) vec4 clamp(vec4 x, float minVal, float maxVal)	Ограничивает значение x границами minVal и maxVal; то есть, возвращает $\min(\max(x, \text{minVal}), \text{maxVal})$. Результат неопределен, если $\text{minVal} > \text{maxVal}$.
float mix(float x, float y, float a) vec2 mix(vec2 x, vec2 y, float a) vec3 mix(vec3 x, vec3 y, float a) vec4 mix(vec4 x, vec4 y, float a) vec2 mix(vec2 x, float y, vec2 a) vec3 mix(vec3 x, float y, vec3 a) vec4 mix(vec4 x, float y, vec4 a) vec2 mix(vec2 x, vec2 y, vec2 a) vec3 mix(vec3 x, vec3 y, vec3 a) vec4 mix(vec4 x, vec4 y, vec4 a)	Возвращает результат линейного смешивания x и y; то есть $x * (1 - a) + y * a$.

Синтаксис	Описание
<pre>float step(float edge, float x) vec2 step(vec2 edge, vec2 x) vec3 step(vec3 edge, vec3 x) vec4 step(vec4 edge, vec4 x) vec2 step(float edge, vec2 x) vec3 step(float edge, vec3 x) vec4 step(float edge, vec4 x)</pre>	Сравнивает два значения и возвращает 0.0, если $x < \text{edge}$, иначе возвращает 1.0.
<pre>float smoothstep(float edge0, float edge1, float x) vec2 smoothstep(vec2 edge0, vec2 edge1, vec2 x) vec3 smoothstep(vec3 edge0, vec3 edge1, vec3 x) vec4 smoothstep(vec4 edge0, vec4 edge1, vec4 x)</pre>	Эрмитова интерполяция. Возвращает 0.0, если $x \leq \text{edge0}$, 1.0, если $x \geq \text{edge1}$, производит сглаживание с применением эрмитовой интерполяции между 0 и 1, когда $\text{edge0} < x < \text{edge1}$. Действует эквивалентно следующему фрагменту: <pre>//genType - float, vec2, vec3 или vec4 genType t; t = clamp((x - edge0) / (edge1 - edge0), 0, 1); return t * t * (3 - 2 * t);</pre> Результат неопределен, если $\text{edge0} \geq \text{edge1}$.

Функции, следующие далее, определяют, какие компоненты их параметров будут использоваться, в зависимости от назначения функции.

Геометрические функции

Синтаксис	Описание
<pre>float length(float x) float length(vec2 x) float length(vec3 x) float length(vec4 x)</pre>	Возвращает длину вектора x .
<pre>float distance(float p0, float p1) float distance(vec2 p0, vec2 p1) float distance(vec3 p0, vec3 p1) float distance(vec4 p0, vec4 p1)</pre>	Возвращает расстояние между $p0$ и $p1$; то есть, $\text{length}(p0 - p1)$.
<pre>float dot(float x, float y) float dot(vec2 x, vec2 y) float dot(vec3 x, vec3 y) float dot(vec4 x, vec4 y)</pre>	Возвращает скалярное произведение x и y; для случая vec3: $x[0] * y[0] + x[1] * y[1] + x[2] * y[2].$
<pre>vec3 cross(vec3 x, vec3 y)</pre>	Возвращает векторное произведение x и y, для случая vec3: $\begin{aligned} \text{result}[0] &= x[1] * y[2] - y[1] * x[2] \\ \text{result}[1] &= x[2] * y[0] - y[2] * x[0] \\ \text{result}[2] &= x[0] * y[1] - y[0] * x[1] \end{aligned}$

Синтаксис	Описание
<pre>float normalize(float x) vec2 normalize(vec2 x) vec3 normalize(vec3 x) vec4 normalize(vec4 x)</pre>	Возвращает вектор, направление которого совпадает с направлением вектора <i>x</i> , но имеет длину, равную 1; то есть $x / \text{length}(x)$.
<pre>float faceforward(float N, float I, float Nref) vec2 faceforward(vec2 N, vec2 I, vec2 Nref) vec3 faceforward(vec3 N, vec3 I, vec3 Nref) vec4 faceforward(vec4 N, vec4 I, vec4 Nref)</pre>	Выполняет обращение нормали. Корректирует вектор <i>N</i> в соответствии с вектором падения <i>I</i> и базисного вектора <i>Nref</i> . Если $\text{dot}(Nref, I) < 0$, вернет <i>N</i> , иначе вернет $-N$.
<pre>float reflect(float I, float N) vec2 reflect(vec2 I, vec2 N) vec3 reflect(vec3 I, vec3 N) vec4 reflect(vec4 I, vec4 N)</pre>	Вычисляет вектор отражения. Для вектора падения <i>I</i> и ориентации поверхности <i>N</i> вернет направление отражения: $I - 2 * \text{dot}(N, I) * N$ Вектор <i>N</i> должен быть нормализован.
<pre>float refract(float I, float N, float eta) vec2 refract(vec2 I, vec2 N, float eta) vec3 refract(vec3 I, vec3 N, float eta) vec4 refract(vec4 I, vec4 N, float eta)</pre>	Вычисляет отклонение от направления на источник света, вызванное характеристиками преломления среды, путем вычисления вектора падения с использованием коэффициентов преломления. Для вектора падения <i>I</i> , нормали <i>N</i> и коэффициента преломления <i>eta</i> вернет вектор, являющийся результатом следующих вычислений: <pre>k = 1.0 - eta * eta * (1.0 - dot(N, I) * dot(N, I)) if (k < 0.0) // genTyp - float, vec2, vec3 или vec4 return genType(0.0) else return eta * I - (eta * dot(N, I) + sqrt(k)) * N</pre>
	Вектор падения <i>I</i> и вектор нормали <i>N</i> должны быть нормализованы.

Матричные функции

Синтаксис	Описание
<pre>mat2 matrixCompMult(mat2 x, mat2 y) mat3 matrixCompMult(mat3 x, mat3 y) mat4 matrixCompMult(mat4 x, mat4 y)</pre>	Умножает матрицу <i>x</i> на матрицу <i>y</i> , покомпонентно; то есть, если $\text{result} = \text{matrixCompMatrix}(x, y)$, тогда $\text{result}[i][j] = x[i][j] * y[i][j]$.

Векторные функции

Синтаксис	Описание
bvec2 lessThan(vec2 x, vec2 y) bvec3 lessThan(vec3 x, vec3 y) bvec4 lessThan(vec4 x, vec4 y) bvec2 lessThan(ivec2 x, ivec2 y) bvec3 lessThan(ivec3 x, ivec3 y) bvec4 lessThan(ivec4 x, ivec4 y)	Возвращает результат покомпонентного сравнения $x < y$.
bvec2 lessThanEqual(vec2 x, vec2 y) bvec3 lessThanEqual(vec3 x, vec3 y) bvec4 lessThanEqual(vec4 x, vec4 y) bvec2 lessThanEqual(ivec2 x, ivec2 y) bvec3 lessThanEqual(ivec3 x, ivec3 y) bvec4 lessThanEqual(ivec4 x, ivec4 y)	Возвращает результат покомпонентного сравнения $x \leq y$.
bvec2 greaterThan(vec2 x, vec2 y) bvec3 greaterThan(vec3 x, vec3 y) bvec4 greaterThan(vec4 x, vec4 y) bvec2 greaterThan(ivec2 x, ivec2 y) bvec3 greaterThan(ivec3 x, ivec3 y) bvec4 greaterThan(ivec4 x, ivec4 y)	Возвращает результат покомпонентного сравнения $x > y$.
bvec2 greaterThanEqual(vec2 x, vec2 y) bvec3 greaterThanEqual(vec3 x, vec3 y) bvec4 greaterThanEqual(vec4 x, vec4 y) bvec2 greaterThanEqual(ivec2 x, ivec2 y) bvec3 greaterThanEqual(ivec3 x, ivec3 y) bvec4 greaterThanEqual(ivec4 x, ivec4 y)	Возвращает результат покомпонентного сравнения $x \geq y$.
bvec2 equal(vec2 x, vec2 y) bvec3 equal(vec3 x, vec3 y) bvec4 equal(vec4 x, vec4 y) bvec2 equal(ivec2 x, ivec2 y) bvec3 equal(ivec3 x, ivec3 y) bvec4 equal(ivec4 x, ivec4 y)	Возвращает результат покомпонентного сравнения $x == y$.
bvec2 notEqual(vec2 x, vec2 y) bvec3 notEqual(vec3 x, vec3 y) bvec4 notEqual(vec4 x, vec4 y) bvec2 notEqual(ivec2 x, ivec2 y) bvec3 notEqual(ivec3 x, ivec3 y) bvec4 notEqual(ivec4 x, ivec4 y)	Возвращает результат покомпонентного сравнения $x != y$.
bool any(bvec2 x) bool any(bvec3 x) bool any(bvec4 x)	Возвращает true, если хотя бы один компонент x имеет значение true.
bool all(bvec2 x) bool all(bvec3 x) bool all(bvec4 x)	Возвращает true, только если все компоненты x имеют значение true.
bvec2 not(bvec2 x) bvec3 not(bvec3 x) bvec4 not(bvec4 x)	Возвращает логические дополнения для компонентов x.

Функции для работы с текстурами

Синтаксис	Описание
<pre>vec4 texture2D(sampler2D sampler, vec2 coord) vec4 texture2D(sampler2D sampler, vec2 coord, float bias) vec4 texture2DProj(sampler2D sampler, vec3 coord) vec4 texture2DProj(sampler2D sampler, vec3 coord, float bias) vec4 texture2DProj(sampler2D sampler, vec4 coord) vec4 texture2DProj(sampler2D sampler, vec4 coord, float bias) vec4 texture2DLod(sampler2D sampler, vec2 coord, float lod) vec4 texture2DProjLod(sampler2D sampler, vec3 coord, float lod) vec4 texture2DProjLod(sampler2D sampler, vec4 coord, float lod)</pre>	Читает значение текселя с координатами <code>coord</code> в 2-мерной текстуре, связанной с семплером <code>sampler</code>. Проективные версии (<code>Proj</code>) делят координаты (<code>coord.s</code>, <code>coord.t</code>) на последний компонент <code>coord</code>. Третий компонент <code>coord</code> игнорируется для аргумента <code>vec4 coord</code>. Параметр <code>bias</code> доступен только во фрагментных шейдерах. Он определяет значение для добавления к текущему значению <code>lod</code>, когда с семплером связана MIP-текстура.
<pre>vec4 textureCube(samplerCube sampler, vec3 coord) vec4 textureCube(samplerCube sampler, vec3 coord, float bias) vec4 textureCubeLod(samplerCube sampler, vec3 coord, float lod)</pre>	Читает значение текселя с координатами <code>coord</code> в кубической текстуре, связанной с семплером <code>sampler</code>. Направление <code>coord</code> используется для выбора лицевой стороны.



ПРИЛОЖЕНИЕ С.

Матрицы проекций

Матрица ортогональной проекции

Вызов метода `Matrix4.setOrtho(left, right, bottom, top, near, far)` создаст следующую матрицу:

$$\begin{bmatrix} \frac{2}{right - left} & 0 & 0 & -\frac{right + left}{right - left} \\ 0 & \frac{2}{top - bottom} & 0 & -\frac{top + bottom}{top - bottom} \\ 0 & 0 & -\frac{2}{far - near} & -\frac{far + near}{far - near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Матрица перспективной проекции

Вызов метода `Matrix4.setPerspective(fov, aspect, near, far)` создаст следующую матрицу:

$$\begin{bmatrix} \frac{1}{aspect * \tan(\frac{fov}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{fov}{2})} & 0 & 0 \\ 0 & 0 & -\frac{far + near}{far - near} & -\frac{2 * far * near}{far - near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$



ПРИЛОЖЕНИЕ D.

WebGL/OpenGL:

лево- или правосторонняя система координат?

В главе 2, «Первые шаги в WebGL», мы говорили, что система WebGL использует правостороннюю систему координат. Однако, вам наверняка попадутся книги, статьи или другие источники информации в Веб, которые будут противоречить этому утверждению. В данном приложении вы узнаете, какие системы координат в действительности использует WebGL и что получится, если попробовать нарисовать что-либо, используя WebGL с настройками по умолчанию. Спецификация WebGL основана на спецификации OpenGL, поэтому все, что вы здесь узнаете, в равной степени применимо и к OpenGL. Это приложение следует читать после знакомства с главой 7, «Вперед, в трехмерный мир», потому что здесь мы не раз будем ссылаться на примеры программ в этой главе.

Для начала обратимся к «источнику всех знаний»: к оригиналу спецификации, точнее к спецификации OpenGL ES 2.0, которая послужила основой для WebGL, и была опубликована консорциумом Khronos Group.¹ В приложении В это спецификации отмечается:

7. Графическая библиотека не привязана ни к левосторонней, ни к правосторонней системе координат.

Если дело действительно обстоит так и WebGL не привязана к какой-то определенной системе координат, тогда почему так много книг, и эта книга в том числе, описывают WebGL, как использующую правостороннюю систему координат? В действительности, это всего лишь соглашение. Начиная разработку своего приложения, вам необходимо решить, какая система координат будет использоваться, и придерживаться этого решения. Это верно не только для ваших приложений, но и для библиотек, которые создаются другими разработчиками, с целью помочь другим программистам использовать WebGL (и OpenGL). Многие из этих библиотек следуют за соглашением об использовании правосторонней системы координат, поэтому со временем данное соглашение стало общепринятым и пре-

¹ http://www.khronos.org/registry/gles/specs/2.0/es_cm_spec_2.0.24.pdf

вратилось в синоним WebGL (и OpenGL), заставляя многих думать, что WebGL (и OpenGL) привязана к правосторонней системе координат.

И что же здесь не так? Если все будут следовать за одними и теми же соглашениями, это не должно вызывать никаких проблем. Да, это так, но возникает одна сложность, потому что WebGL (и OpenGL) иногда требуют, чтобы графическая библиотека выполняла операции без учета направленности системы координат – с настройками по умолчанию, если хотите, – а настройки по умолчанию не всегда оказываются правосторонними!

В этом приложении мы посмотрим, как действует WebGL с настройками по умолчанию, чтобы вы могли четко понять суть проблемы и как избежать неприятностей в своих приложениях.

Для начала напомним программу `CoordinateSystem`, которая будет служить нам полигоном для экспериментов. С помощью этой программы мы вернемся сначала к самым основам – приемам рисования треугольников – и затем будем наращивать ее возможности, исследуя, как WebGL рисует множество объектов. В данной программе перед нами стоит цель – нарисовать синий треугольник в позиции -0.1 на оси Z и красный треугольник в позиции -0.5 на оси Z . На рис. D.1 показаны треугольники, которые требуется нарисовать, их координаты Z и цвета.

Как будет показано далее в этом приложении, чтобы достигнуть нашей относительно скромной цели, нам потребуется организовать взаимодействие множества механизмов, включая механизм простого рисования, механизм удаления скрытых поверхностей и механизм отображения видимого объема. Без должной настройки всех трех механизмов мы получим довольно неожиданные результаты, которые могут привести к путанице между лево- и правосторонними системами координат.

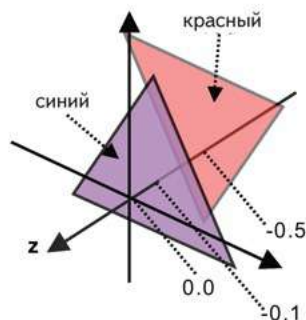


Рис. D.1. Треугольники, используемые в качестве примеров в этом приложении, и их цвета

Пример программы `CoordinateSystem.js`

В листинге D.1 представлен исходный код программы `CoordinateSystem.js`. Мы убрали из листинга код обработки ошибок и некоторые комментарии ради экономии места, однако в загружаемом пакете программ вы найдете полный исходный код.

Листинг D.1. `CoordinateSystem.js`

```
1 // CoordinateSystem.js
2 // Вершинный шейдер
```

```

3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'attribute vec4 a_Color;\n' +
6   'varying vec4 v_Color;\n' +
7   'void main() {\n' +
8   '   gl_Position = a_Position;\n' +
9   '   v_Color = a_Color;\n' +
10  '}\n';
11
12 // Фрагментный шейдер
13 var FSHADER_SOURCE =
14   '#ifdef GL_ES\n' +
15   'precision mediump float;\n' +
16   '#endif\n' +
17   'varying vec4 v_Color;\n' +
18   'void main() {\n' +
19   '   gl_FragColor = v_Color;\n' +
20   '}\n';
21
22 function main() {
23   var canvas = document.getElementById('webgl'); // Получить <canvas>
24   var gl = getWebGLContext(canvas); // Получить контекст WebGL
25   initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE); // Инициализировать шейдеры
26   var n = initVertexBuffers(gl); // Определить координаты вершин и цвета
27
28   gl.clearColor(0.0, 0.0, 0.0, 1.0); // Определить цвет очистки
29   gl.clear(gl.COLOR_BUFFER_BIT); // Очистить <canvas>
30   gl.drawArrays(gl.TRIANGLES, 0, n); // Нарисовать треугольники
31 }
32
33 function initVertexBuffers(gl) {
34   var pc = new Float32Array([ // Координаты вершин и цвета
35     0.0, 0.5, -0.1, 0.0, 0.0, 1.0, // Синий треугольник, ближний
36     -0.5, -0.5, -0.1, 0.0, 0.0, 1.0,
37     0.5, -0.5, -0.1, 1.0, 1.0, 0.0,
38
39     0.5, 0.4, -0.5, 1.0, 1.0, 0.0, // Красный треугольник, дальний
40     -0.5, 0.4, -0.5, 1.0, 0.0, 0.0,
41     0.0, -0.6, -0.5, 1.0, 0.0, 0.0,
42   ]);
43   var numVertex = 3; var numColor = 3; var n = 6;
44
45   // Создать буферный объект и записать в него данные
46   var pcbuffer = gl.createBuffer();
47   gl.bindBuffer(gl.ARRAY_BUFFER, pcbuffer);
48   gl.bufferData(gl.ARRAY_BUFFER, pc, gl.STATIC_DRAW);
49
50   var FSIZE = pc.BYTES_PER_ELEMENT; // Число байтов
51   var STRIDE = numVertex + numColor; // Вычислить шаг
52
53   // Присвоить координаты вершины переменной-атрибуту
54   var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
55   gl.vertexAttribPointer(a_Position, numVertex, gl.FLOAT, false, FSIZE *
56     ➡ STRIDE, 0);
57   gl.enableVertexAttribArray(a_Position);

```

```
57
58 // Присвоить цвета переменной-атрибуту
59 var a_Color = gl.getAttribLocation(gl.program, 'a_Color');
60 gl.vertexAttribPointer(a_Color, numColor, gl.FLOAT, false, FSIZE *
    ↳ STRIDE, FSIZE * numVertex);
61 gl.enableVertexAttribArray(a_Color);
62
63 return n;
64 }
```

Если запустить эту программу, она создаст изображение, показанное на рис. D.2. На черно-белом рисунке это сложно увидеть (не забывайте, что вы можете запустить программу в браузере на веб-сайте книги), поэтому сразу скажу, что красный треугольник нарисован перед синим. Это противоречит нашим ожиданиям, потому что согласно определениям координат вершин в строках с 32 по 42 синий треугольник должен быть нарисован перед красным.

Однако, как описывалось в главе 7, здесь нет никакой ошибки. Так получилось просто потому, что WebGL сначала нарисовала синий треугольник, потому что его координаты определены первыми, а потом красный, поверх синего. Это немного напоминает рисование красками; нижний слой краски будет закрыт слоем краски, наносимым сверху.

Многие, только начинающие осваивать WebGL, находят такое поведение противоречащим здравому смыслу. Поскольку WebGL – это система для создания трехмерной графики, можно было бы ожидать, что она «все сделает правильно» и нарисует красный треугольник за синим. Однако, по умолчанию WebGL рисует фигуры в том порядке, в каком они определены и не учитывает их размещение вдоль оси Z. Если вам нужно, чтобы WebGL действительно «все сделала правильно», следует включить механизм Удаления Скрытых Поверхностей, обсуждавшийся в главе 7. Как там было показано, механизм Удаления Скрытых Поверхностей помогает системе WebGL проанализировать трехмерную сцену и удалить поверхности, которые действительно не видны с данной точки наблюдения. В нашем случае этот механизм должен помочь ликвидировать проблему отображения красного треугольника в трехмерной сцене, скрыв большую его часть за синим треугольником.



Рис. D.2. CoordinateSystem

Удаление скрытых поверхностей и усеченная система координат

Давайте включим механизм удаления скрытых поверхностей в нашей программе и посмотрим, что из этого получится. Для этого нужно вызвать функцию

`gl.enable(gl.DEPTH_TEST)`, очистить буфер глубины и затем нарисовать треугольники. Сначала добавим следующий код в строку 27.

```
27 gl.enable(gl.DEPTH_TEST);
```

Затем изменим строку 29, как показано ниже:

```
29 gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

Теперь мы вправе ожидать, что проблема решена, и программа нарисует красный треугольник за синим. Однако, если запустить измененную программу, вы обнаружите, что она по-прежнему рисует красный треугольник поверх синего, как показано на рис. D.3.

Неожиданно, но это часть таинственности, окружающей направленность системы координат WebGL. Мы правильно реализовали нашу программу, исходя из предположения, что WebGL использует правостороннюю систему координат. Но, похоже, что в системе координат WebGL точка с координатой -0.5 на оси Z находится перед точкой с координатой -0.1 , либо в действительности WebGL использует левостороннюю систему координат, в которой положительное направление оси Z простирается от пользователя за плоскость экрана (рис. D.4).



Рис. D.3. CoordinateSystem с включенным механизмом удаления скрытых поверхностей

Усеченная система координат и видимый объем

Итак, наше приложение следует за соглашением об использовании правосторонней системы координат, но оно наглядно демонстрирует, что в действительности используется левосторонняя система координат. Как это объяснить? Фактически, когда включается механизм удаления скрытых поверхностей, в действие вступает **усеченная система координат** (clip coordinate system) (см. рис. G.5 в приложении G), которая сама по себе является левосторонней.

В WebGL (OpenGL) удаление скрытых поверхностей выполняется с использованием значения `gl_Position` — координат, которые определяются вершинным шейдером. Как можно видеть в ли-

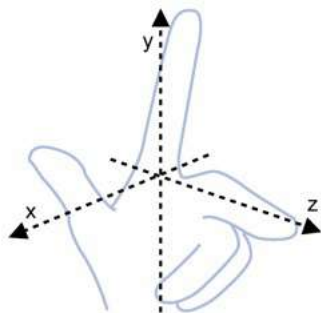


Рис. D.4. Левосторонняя система координат

стинге D.1 (строка 8), вершинный шейдер просто присваивает `a_Position` переменной `gl_Position`. Это означает, что координата Z красного треугольника передается как -0.5 , а координата Z синего треугольника передается как -0.1 в усеченной системе координат (левосторонней). Как вы уже знаете, положительное направление оси Z в левосторонней системе координат направлено за плоскость экрана, поэтому меньшая координата Z (-0.5) оказывается ближе к наблюдателю, чем большая координата (-0.1). Соответственно система WebGL все сделала правильно, нарисовав красный треугольник перед синим.

Совершенно очевидно, что такое поведение противоречит описанию в главе 3 (где утверждалось, что WebGL использует правостороннюю систему координат). Так как же нам добиться нашей цели и отобразить красный треугольник за синим, и как полученные нами результаты соотносятся с поведением системы WebGL по умолчанию? До сих пор в этой программе мы никак не учитывали видимый объем, который обязательно нужно настроить при использовании механизма удаления скрытых поверхностей. Настройка заключается в определении ближней и дальней плоскостей отсечения, причем ближняя плоскость должна находиться перед дальней (то есть, должно выполняться условие $near < far$). Здесь *near* и *far* – это расстояния от точки наблюдения вдоль линии взгляда и могут иметь любые значения. Соответственно в качестве расстояния *far* можно указать меньшее значение, чем значение *near*, или даже использовать отрицательное значение. (Отрицательные значения интерпретируются как расстояние от точки наблюдения в направлении, противоположном направлению взгляда.) Очевидно, что значения, выбираемые для *near* и *far*, зависят от используемой системы координат – лево- или правосторонней.

Но, вернемся к примеру программы и попробуем правильно настроить видимый объем, чтобы добиться желаемого результата от механизма удаления скрытых поверхностей. В листинге D.2 приводятся только отличия от `CoordinateSystem.js`.

Листинг D.2. `CoordinateSystem_viewVolume.js`

```
1 // CoordinateSystem_viewVolume.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'attribute vec4 a_Color;\n' +
6   'uniform mat4 u_MvpMatrix;\n' +
7   'varying vec4 v_Color;\n' +
8   'void main() {\n' +
9     '  gl_Position = u_MvpMatrix * a_Position;\n' +
10    '  v_Color = a_Color;\n' +
11    '}\n';
12
13 ...
23 function main() {
24   ...
29   gl.enable(gl.DEPTH_TEST); // Включить удаление скрытых поверхностей
30   gl.clearColor(0.0, 0.0, 0.0, 1.0); // Определить цвет очистки
31   // Получить ссылку на u_MvpMatrix
```

```

32 var u_MvpMatrix = gl.getUniformLocation(gl.program, 'u_MvpMatrix');
33
34 var mvpMatrix = new Matrix4();
35 mvpMatrix.setOrtho(-1, 1, -1, 1, 0, 1); // Определить видимый объем
36 // Передать матрицу вида в u_MvpMatrix
37 gl.uniformMatrix4fv(u_MvpMatrix, false, mvpMatrix.elements);
38
39 gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
40 gl.drawArrays(gl.TRIANGLES, 0, n); // Нарисовать треугольники
41 }

```

Если запустить эту программу, на экране появится изображение, показанное на рис. D.5, где синий треугольник нарисован перед красным.



Рис. D.5. CoordinateSystem_viewVolume

Важнейшим отличием этой версии программы является добавление в вершинный шейдер `uniform`-переменной (`u_MvpMatrix`) для передачи матрицы вида. Она умножается на вектор координат `a_Position` и результат присваивается переменной `gl_Position`. Хотя в данном примере для определения видимого объема мы использовали метод `setOrtho()`, метод `setPerspective()` даст тот же результат.

Теперь все правильно?

Давайте сравним вершинные шейдеры в программах `CoordinateSystem.js` и `CoordinateSystem_viewVolume.js`.

Строка 8 в `CoordinateSystem.js`:

```
8 ' gl_Position = a_Position;\n' +
```

превратилась в строку 9 в `CoordinateSystem_viewVolume.js`:

```
9 ' gl_Position = u_MvpMatrix * a_Position;\n' +
```

Как видите, в программе `CoordinateSystem_viewVolume.js`, которая отображает треугольники в требуемом нам порядке, матрица преобразований (в дан-

ном случае матрица вида) умножается на координаты вершины. Чтобы понять суть этой операции, посмотрим, как можно переписать строку 8 в программе `CoordinateSystem.js` в форме `<матрица>*<координаты вершины>`, по аналогии со строкой 9 в программе `CoordinateSystem_viewVolume.js`.

Строка 8 присваивает координаты вершины (`a_Position`) переменной `gl_Position` без каких-либо промежуточных преобразований. Чтобы гарантировать, что добавление операции умножения матрицы не окажет никакого влияния, `<матрица>` должна иметь следующие элементы (то есть, она должна быть единичной матрицей):

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

То есть, строка 8 в `CoordinateSystem.js` фактически дает тот же результат, что и передача единичной матрицы в `u_MvpMatrix`, в строке 9 в `CoordinateSystem_viewVolume.js`. По сути, эта матрица управляет поведением по умолчанию системы WebGL.

Чтобы лучше понять это поведение, давайте посмотрим, что получится, если в качестве матрицы проекции передать единичную матрицу. Для этого воспользуемся матрицей из приложения C (см. рис. D.6) и найдем в единичной матрице значения *left*, *right*, *top*, *bottom*, *near* и *far*.

$$\begin{bmatrix} \frac{2}{right - left} & 0 & 0 & -\frac{right + left}{right - left} \\ 0 & \frac{2}{top - bottom} & 0 & -\frac{top + bottom}{top - bottom} \\ 0 & 0 & -\frac{2}{far - near} & -\frac{far + near}{far - near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Рис. D.6. Матрица проекции, генерируемая методом `setOrtho()`

В данном случае $right - left = 2$ и $right + left = 0$, что дает в результате $left = -1$ и $right = 1$. Аналогично, $far - near = -2$ и $far + near = 0$, что дает $near = 1$ и $far = -1$. То есть:

```
left = -1, right = 1, bottom = -1, top = 1, near = 1, far = -1
```

Передадим эти параметры в вызов `setOrtho()`:

```
mvpMatrix.setOrtho(-1, 1, -1, 1, 1, -1);
```

Как видите, значение *near* здесь больше значения *far*, а это означает, что дальняя плоскость отсечения находится перед ближней плоскостью отсечения, если смотреть в направлении взгляда (см. рис. D.7).

Подобное явление наблюдается и при определении видимого объема вручную, если задать $near > far$. То есть, когда видимый объем определяется таким способом, WebGL (OpenGL) следует за соглашением о правосторонней системе координат.

Теперь рассмотрим матрицу, представляющую видимый объем, в котором все объекты отображаются правильно:

```
mvpMatrix.setOrtho(-1, 1, -1, 1, -1, 1);
```

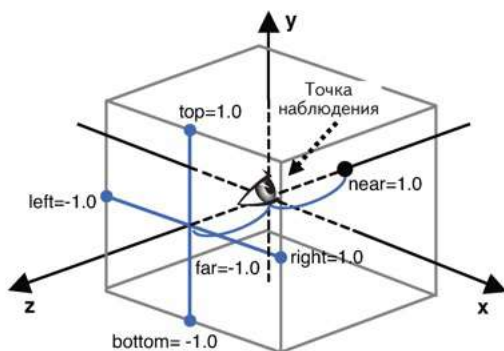


Рис. D.7. Видимый объем, созданный с применением единичной матрицы

Этот вызов метода сгенерирует следующую матрицу:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

В ней без труда можно узнать матрицу масштабирования, описанную в главе 4, «Дополнительные преобразования и простая анимация». Эта матрица генерируется вызовом `setScale(1, 1, -1)`. Обратите внимание на масштабный множитель для оси Z, который равен -1 , это означает изменение направления оси Z на обратное. То есть, эта матрица преобразует привычную для нас правостороннюю систему координат, используемую в этой книге (и предполагаемую большинством библиотек для WebGL), в левостороннюю, используемую в усеченной системе координат.

В заключение

Итак, обратившись к спецификации WebGL, мы выяснили, что система WebGL не связана с какой-то конкретной, лево- или правосторонней системой координат. Мы узнали, что многие библиотеки и приложения для WebGL следуют за соглашением об использовании правосторонней системы координат. Когда поведение WebGL по умолчанию противоречит этому соглашению (например, при использовании усеченного пространства, где используется левосторонняя система координат), эту проблему можно устранить программно, изменив знак, к примеру, координат Z. Это позволит продолжать следовать за соглашением об использовании правосторонней системы координат. Однако, как отмечалось выше, это всего лишь соглашение, одно из многих, которым следуют люди, но также одно из тех, что может сбивать с толку, если не знать, как действует WebGL по умолчанию и как это исправить.

ПРИЛОЖЕНИЕ Е.

Транспонированная обратная матрица

Транспонированная обратная матрица, с которой мы познакомились в главе 8, «Освещение объектов», – это матрица, обратная по отношению к заданной, и затем транспонированная. Как показано на рис. Е.1, направление вектора нормали к поверхности объекта изменяется, в зависимости от типа преобразований координат. Однако, если использовать транспонированную обратную матрицу, эти вычисления можно безопасно игнорировать.

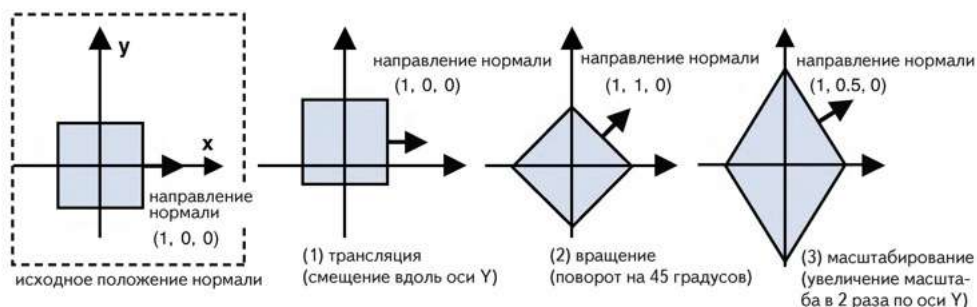


Рис. Е.1. Направление вектора нормали изменяется, в зависимости от преобразований координат

В главе 8 мы видели, как использовать транспонированную обратную матрицу модели для преобразования нормалей. Однако, в некоторых ситуациях направление векторов нормалей можно так же определить непосредственно по исходной матрице модели. Примером может служить операция вращения – в этом случае направление вектора нормали можно определить, умножив вектор нормали на матрицу вращения. Выбор матрицы (исходной матрицы модели или транспонированной обратной ее матрицы) для вычисления направления вектора нормали зависит от того, какие преобразования координат она описывает (трансляцию, вращение и масштабирование).

Если матрица модели включает трансляцию (перемещение) и вы попытаетесь умножить вектор нормали на нее, направление вектора изменится. Например,

если нормаль (1, 0, 0) умножить на матрицу, описывающую перемещение вдоль оси X на 2.0 единицы, получится вектор нормали (1, 2, 0). Этой проблемы можно избежать, если использовать подматрицу 3×3, извлеченную из левого верхнего угла матрицы модели 4×4. Например:

```
attribute vec4 a_Normal;    // нормаль
uniform mat4 u_ModelMatrix; // матрица модели
void main() {
    ...
    vec3 normal = normalize(mat3(u_ModelMatrix) * a_Normal.xyz);
    ...
}
```

Значения, находящиеся в самом правом столбце, определяют величину смещения, описываемого матрицей трансляции, как показано на рис. Е.2.

подматрица 3×3

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

(подматрица 3×3 выделена синим цветом)

Матрица преобразования (матрица трансляции 4×4)

Рис. Е.2. Матрица преобразования и ее подматрица 3×3

Так как эта подматрица включает также компоненты, описывающие поворот и масштабирование, рассмотрим вращение и масштабирование каждый случай в отдельности:

- **когда требуется выполнить только поворот:** можно использовать подматрицу 3×3 из исходной матрицы модели, при этом, даже если вектор нормали был нормализован, преобразованный вектор нормали получится ненормализованным;
- **когда требуется выполнить масштабирование (однородное масштабирование):** можно использовать подматрицу 3×3 из исходной матрицы модели, при этом преобразованный вектор нормали получится нормализованным;
- **когда требуется выполнить масштабирование (неоднородное масштабирование):** требуется использовать транспонированную обратную матрицу модели, при этом преобразованный вектор нормали получится нормализованным.

Во втором случае, когда требуется выполнить однородное масштабирование, предполагается, что по всем осям – X, Y и Z – масштабирование будет выполнено с применением одного и того же масштабного множителя. Например, если предполагается выполнить масштабирование по всем трем осям с коэффициентом 2.0, это значение следует передать во всех аргументах в вызов `Matrix4.scale()`: `Matrix4.scale(2.0, 2.0, 2.0)`. В этой ситуации, несмотря на изменение разме-

ров объекта, его форма останется прежней. Иная ситуация складывается, при неоднородном масштабировании, когда масштабирование выполняется по разным осям с разными коэффициентами. Например, если масштабирование требуется выполнить только вдоль оси Y , следует использовать вызов `Matrix4.scale(1.0, 2.0, 1.0)`.

В третьем случае требуется найти транспонированную обратную матрицу модели, потому что, если масштабирование выполняется неоднородно, направление вектора нормали будет найдено неверно, если умножить его на исходную матрицу модели. Эта ситуация изображена на рис. Е.3.

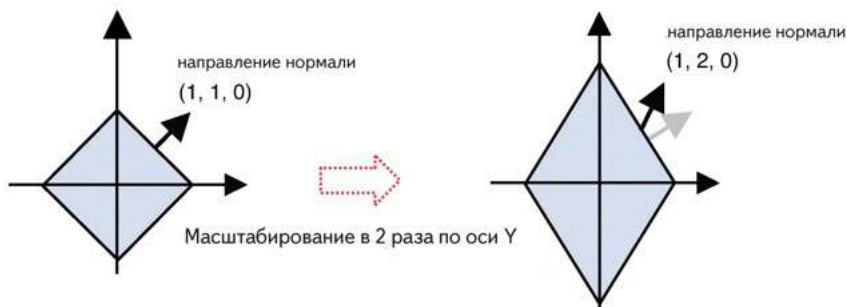


Рис. Е.3. Простое умножение вектора нормали на исходную матрицу модели даст неверный результат

Масштабирование объекта (на рис. Е.3 слева) вдоль оси Y с коэффициентом 2.0 приведет к изменению его формы (на рис. Е.3 справа). Если в этом случае, чтобы определить новое направление нормали после преобразования, умножить матрицу модели на нормаль $(1, 1, 0)$ исходного объекта, получится нормаль $(1, 2, 0)$, которая не образует угол 90 градусов с плоскостью.

Чтобы решить эту проблему, необходимо прибегнуть к математическому аппарату. Далее мы будем называть матрицу модели M , исходный вектор нормали n , матрицу преобразования M' (она преобразует n без изменения направления) и перпендикуляр к n как s . Кроме того, определим n' и s' , как показано в формулах Е.1 и Е.2:

Формула Е.1.

$$n' = M' \times n$$

Формула Е.2.

$$s' = M \times s$$

Взгляните на рис. Е.4.

Здесь требуется вычислить матрицу M' такую, чтобы векторы n' и s' остались перпендикулярными. Если два вектора перпендикулярны, их скалярное произведение равно 0. Обозначив операцию скалярного произведения как « $\cdot$ », мы можем записать следующее тождество:

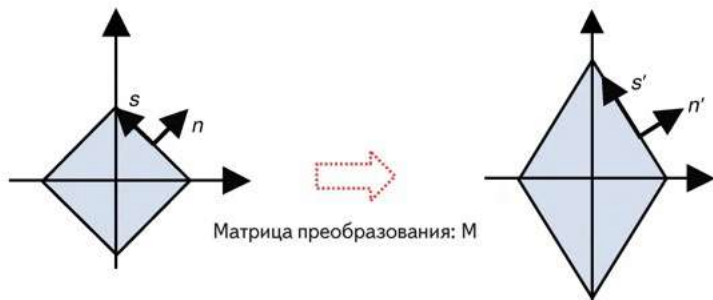


Рис. Е.4. Взаимосвязь между n и s , а также между n' и s'

Формула Е.3.

$$n' \cdot s' = 0$$

Теперь перепишем его, подставив формулы Е.1 и Е.2 (M^T – это транспонированная матрица M):

$$\begin{aligned} (M' \times n) \cdot (M \times s) &= 0 \\ (M' \times n)^T \times (M \times s) &= 0 \quad A \cdot B = A^T \times B \\ n^T \times M'^T \times M \times s &= 0 \quad (A \times B)^T = B^T \times A^T \end{aligned}$$

Так как n и s перпендикулярны, их скалярное произведение также равно 0 ($n \cdot s = 0$). Поэтому, как уже отмечалось, $A \cdot B = A^T \times B$, соответственно, подставив n вместо A и s вместо B , мы получим $n \cdot s = n^T \times s = 0$. Сравнивая это выражение с формулой Е.3, если целью произведения $M'^T \times M$ между n^T и s является единичная матрица (I), тогда его можно записать так:

$$M'^T \times M = I$$

Решив это уравнение, мы получаем следующий результат (M^{-1} – это обратная матрица по отношению к матрице M):

$$M' = (M^{-1})^T$$

Отсюда следует, что M' – это результат транспонирования обратной матрицы M или, говоря другими словами, транспонированная обратная матрица M . Так как матрица M может описывать преобразования, включая случаи (1), (2) и (3), перечисленные выше, если найти транспонированную обратную матрицу M и умножить ее на вектор нормали, получится правильный вектор нормали после преобразования. То есть, мы нашли решение задачи преобразования вектора нормали.

Очевидно, что вычисление транспонированной обратной матрицы требует времени, но, если вы уверены, что матрица преобразования описывает случай (1) или (2), можно просто использовать подматрицу 3×3 , уменьшив время, затрачиваемое на вычисления.

ПРИЛОЖЕНИЕ F.

Загрузка шейдеров из файлов

Во всех примерах программ в этой книге использовались встроенные шейдеры, включенные непосредственно в программный код на JavaScript, что улучшает читаемость примеров программ, но осложняет создание и сопровождение самих шейдеров.

Однако имеется альтернативная возможность загружать исходный код шейдеров из файлов, используя те же методы, что описывались в разделе «Загрузка и отображение трехмерных моделей» в главе 10, «Продвинутые приемы». Чтобы понять, как это делается, давайте изменим программу `ColoredTriangle` из главы 5, «Цвет и текстура», добавив в нее поддержку загрузки шейдеров из файла. Назовем новую программу `LoadShaderFromFiles`. Ее исходный код представлен в листинге F.1.

Листинг F.1. `LoadShaderFromFiles`

```
1 // LoadShaderFromFiles.js based on ColoredTriangle.js
2 // Вершинный шейдер
3 var VSHADER_SOURCE = null;
4 // Фрагментный шейдер
5 var FSHADER_SOURCE = null;
6
7 function main() {
8     // Получить ссылку на элемент <canvas>
9     var canvas = document.getElementById('webgl');
10
11     // Получить контекст отображения WebGL
12     var gl = getWebGLContext(canvas);
13     ...
14     // Загрузить шейдеры из файлов
15     loadShaderFile(gl, 'ColoredTriangle.vert', gl.VERTEX_SHADER);
16     loadShaderFile(gl, 'ColoredTriangle.frag', gl.FRAGMENT_SHADER);
17 }
18
19 function start(gl) {
20     // Инициализировать шейдеры
21     if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
22         ...
23     }
24     gl.drawArrays(gl.TRIANGLES, 0, n);
25 }
```

```
...
88 function loadShaderFile(gl, fileName, shader) {
89   var request = new XMLHttpRequest();
90
91   request.onreadystatechange = function() {
92     if (request.readyState === 4 && request.status !== 404) {
93       onLoadShader(gl, request.responseText, shader);
94     }
95   }
96   request.open('GET', fileName, true);
97   request.send(); // Послать запрос
98 }
99
100 function onLoadShader(gl, fileString, type) {
101   if (type == gl.VERTEX_SHADER) { // Загружен вершинный шейдер
102     VSHADER_SOURCE = fileString;
103   } else
104     if (type == gl.FRAGMENT_SHADER) { // Загружен фрагментный шейдер
105       FSHADER_SOURCE = fileString;
106     }
107   // Начать отображение после загрузки обоих шейдеров
108   if (VSHADER_SOURCE && FSHADER_SOURCE) start(gl);
109 }
```

В отличие от программы `ColoredTriangle.js`, эта программа инициализирует переменные `VSHADER_PROGRAM` (строка 3) и `FSHADER_PROGRAM` (строка 5) значением `null`, чтобы загрузить в них исходный код из файлов. Функция `main()` загружает шейдеры в строках 18 и 19, вызывая функцию `loadShaderFile()`. Определение этой функции начинается в строке 88 и во втором аргументе принимает имя файла (URL) с исходным кодом шейдера. Третий аргумент определяет тип шейдера.

Функция `loadShaderFile()` создает запрос типа `XMLHttpRequest`, чтобы получить файл, определяемый параметром `fileName`, и затем регистрирует обработчик события (`onLoadShader()`) в строке 91, чтобы обработать файл после его загрузки. Затем она посылает запрос (строка 97). После получения файла браузер вызовет обработчик `onLoadShader()`. Определение этой функции начинается в строке 100.

Обработчик `onLoadShader()` проверит третий параметр `type` и в зависимости от его значения сохранит строку `fileString` с исходным кодом шейдера в `VSHADER_PROGRAM` или `FSHADER_PROGRAM`. После загрузки обоих шейдеров он вызывает `start(gl)` в строке 108, чтобы нарисовать треугольник с использованием загруженных шейдеров.



ПРИЛОЖЕНИЕ G.

Мировая и локальная системы координат

В главе 7, «Вперед, в трехмерный мир», мы впервые попробовали нарисовать трехмерный объект (куб), написав программу, более похожую на «настоящее» приложение трехмерной графики. Однако в той программе нам потребовалось вручную определить координаты вершин куба и индексы, что оказалось довольно утомительно. Хотя то же самое мы делали на протяжении всей книги, это не означает, что во всех своих приложениях вам придется вручную вносить все необходимые исходные данные. Обычно для этого используются специализированные инструменты трехмерного моделирования. Их применение позволяет создавать сложные трехмерные объекты, манипулируя (объединяя, деформируя, вращая и т. д.) простыми трехмерными фигурами, такими как кубы, цилиндры или сферы. Один из таких инструментов трехмерного моделирования, Blender (www.blender.org), показан на рис. G.1.

Локальная система координат

При создании трехмерной модели необходимо решить, где будет находиться начало координат (то есть, точка с координатами $(0, 0, 0)$). Начало координат можно выбрать так, чтобы проще было конструировать модель, или, напротив, так, чтобы ею было проще управлять в трехмерной сцене. Куб, упоминавшийся в предыдущем разделе, был помещен своим центром в начало координат. Объекты сферической формы, такие как солнце или луна, также обычно создаются так, что их центр совпадает с центром координат.

С другой стороны, при моделировании персонажей компьютерных игр, таких как лягушонок, изображенный на рис. G.1, модели обычно конструируются так, что центр координат находится в районе ног, а ось Y проходит через центр тела. Если такого персонажа поместить в координату $Y = 0$ (то есть, «поставить» его на поверхность земли), он будет выглядеть, как стоящий на земле — не парящим над ней и не погруженным в землю. Если перемещать такого персонажа вдоль оси X и/или Z , у наблюдателя будет создаваться впечатление, что он «ходит» по поверхности. Кроме того, такого персонажа легко повернуть, реализовав вращение

относительно оси Y.



Рис. G.1. Создание трехмерного объекта с помощью инструмента трехмерного моделирования

В таких случаях координаты вершин, составляющих объекты, определяются относительно начала координат. Такая система координат называется **локальной**. Используя инструменты моделирования (такие как Blender), компоненты (координаты вершин, цвета, индексы и т. д.) моделей, созданных таким способом, можно экспортировать в файлы, импортировать их оттуда в объекты буферов и рисовать их с помощью `gl.drawElements()`.

Мировая система координат

Теперь рассмотрим в качестве примера компьютерную игру со множеством персонажей, действующих в едином пространстве. Цель программиста в этом случае состоит в том, чтобы использовать персонажей, изображенных на рис. G.2 справа, в игровом поле, изображенном слева. Все три персонажа и само игровое поле имеют собственные начала координат.

Когда появляется необходимость отобразить персонажи, возникает проблема. Поскольку все персонажи позиционированы относительно начала собственных систем координат, они будут нарисованы в одной точке, поверх друг друга: в точке $(0, 0, 0)$ в системе координат трехмерной сцены (рис. G.3).¹ Такая ситуация нехарактерна для реального мира и определенно нам не хотелось бы, чтобы она

¹ Для наглядности мы немного сместили персонажей относительно друг друга.

возникала в игре.

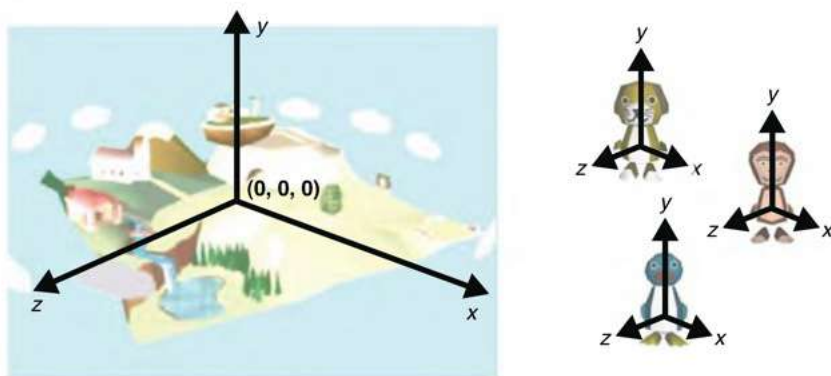


Рис. G.2. Расположение нескольких персонажей в игровом поле

Чтобы исправить эту проблему, нужно изменить местоположение каждого персонажа. Для этого можно воспользоваться преобразованием координат, как рассказывалось в главе 3, «Рисование и преобразование треугольников», и в главе 4, «Дополнительные преобразования и простая анимация». Например, пингвина можно переместить в позицию $(100, 0, 0)$, мартышку – в позицию $(200, 10, 120)$ и собаку – в позицию $(10, 0, 200)$.

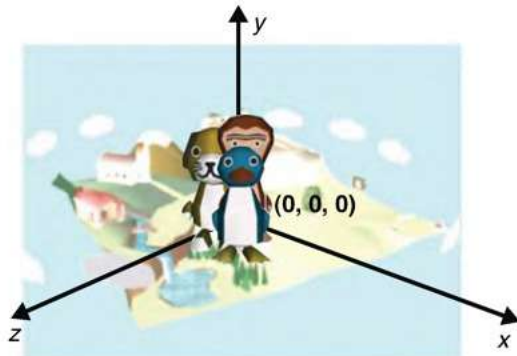


Рис. G.3. Все персонажи оказались в одной точке

Система координат, в которой все персонажи (созданные в локальной системе координат) расставлены по своим местам, называется **мировой**, или **глобальной системой координат**. Соответствующие преобразования моделей называют **мировыми преобразованиями**.

Разумеется, чтобы исключить возможность размещения персонажей в одной точке, можно было бы сразу создать их в мировой системе координат. Например, при создании пингвина в программе Blender, можно было бы сразу указать его координаты $(100, 0, 0)$, тогда, при включении в трехмерную сцену, он сразу оказался бы в точке $(100, 0, 0)$ и нам не потребовалось бы выполнять преобразование коор-

динат. Однако, такой подход создает свои сложности. Например, реализовать вращение такого пингвина вокруг своей оси будет намного труднее, потому что вращение относительно оси Y пришлось реализовать как движение по окружности с радиусом 100. Конечно, можно было бы сначала переместить пингвина в начало координат, повернуть его и затем вернуть на место, но это слишком большой объем работы.

В главе 7 мы уже имели дело с подобным случаем. Используя один ряд треугольников, координаты вершин которых были определены относительно общего центра, программа `PerspectiveView_mvp` рисовала второй ряд треугольников. Ниже показан рисунок, который мы использовали как руководство для создания изображения (см. рис. G.4).

Здесь треугольники, построенные в локальной системе координат, изображены пунктирными линиями, а для описания их перемещений вдоль оси X используется мировая система координат.

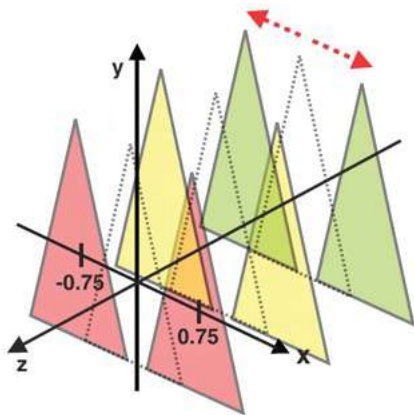


Рис. G.4. Группа треугольников, отображаемая программой `PerspectiveView_mvp`

Преобразования и системы координат

До сих пор мы не различали локальную и мировую системы координат, чтобы вы могли сосредоточить свое внимание на более важных аспектах примеров. Однако, для справки, на рис. G.5 показано, как связаны преобразования и системы координат, и эти взаимосвязи нужно запомнить, так как это пригодится вам по мере того, как вы будете углублять свои познания трехмерной графики и экспериментировать с инструментами трехмерного моделирования.



Рис. G.5. Преобразования и системы координат

ПРИЛОЖЕНИЕ Н.

Настройка поддержки WebGL в веб-браузере

В этом приложении рассказывается, как настроить браузер для корректной работы системы WebGL.

Если графическая карта в вашем компьютере несовместима с WebGL, вы можете увидеть сообщение, как показано на рис. Н.1.

Даже если это случилось, у вас все равно остается шанс заставить WebGL работать в вашем браузере, изменив некоторые настройки:

1. Если вы пользуетесь браузером Chrome, запустите его с параметром `--ignore-gpu-blacklist`. Чтобы определить этот параметр, щелкните правой кнопкой мыши на значке браузера Chrome и выберите пункт **Properties** (Свойства) в контекстном меню. Вы увидите диалог, напоминающий изображение на рис. Н.2. Добавьте указанный параметр в конец командной строки в поле **Target** (Объект). После этого Chrome всегда будет запускаться с данным параметром. Если это помогло решить вашу проблему, оставьте параметр на месте.



Рис. Н.1. При попытке запустить WebGL-приложение браузер вывел сообщение об ошибке

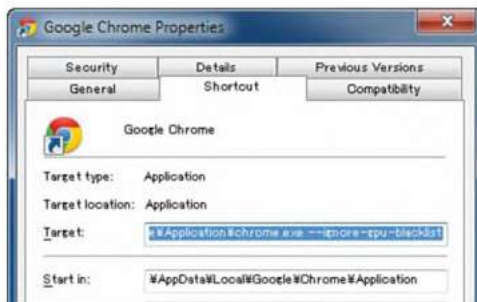


Рис. Н.2. Укажите параметр в диалоге со свойствами браузера Google Chrome

2. Если вы пользуетесь браузером Firefox, введите в адресной строке `about:config`. Браузер выведет предупреждение: «This might void your warranty!» («Будьте осторожны, а то лишитесь гарантии!»). Щелкните на кнопке «I'll be careful, I promise!» («Я обещаю, что буду осторожен!»). Введите `webgl` в текстовое поле с подписью **Search** (Поиск) или **Filter** (Фильтр), после чего Firefox оставит в списке только параметры, имеющие отношение к WebGL (см. рис. Н.3). Дважды щелкните на параметре **webgl.force-enabled** в списке, чтобы его значение изменилось с `false` на `true`. И снова, если это решило вашу проблему, оставьте этот параметр включенным.

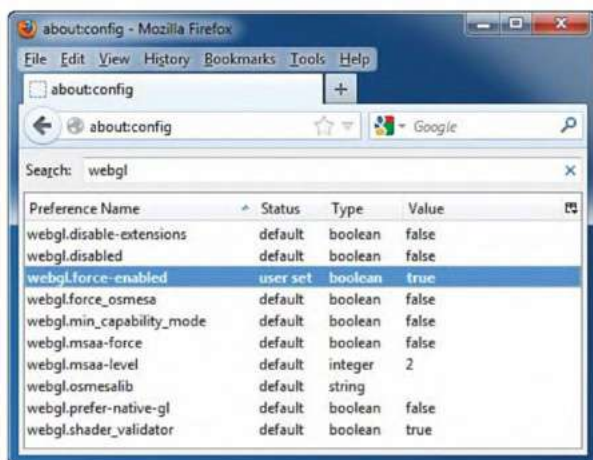


Рис. Н.3. Параметры настройки в Firefox, имеющие отношение к WebGL

Если ничего из вышеперечисленного вам не помогло, тогда вам следует искать другой компьютер, имеющий поддержку WebGL. За дополнительной информацией обращайтесь на вики-страницу консорциума Khronos: www.khronos.org/webgl/wiki.



СЛОВАРЬ ТЕРМИНОВ

С

canvas Элемент HTML5, реализующий поверхность для рисования графики в веб-страницах.

G

GLSL ES OpenGL ES Shading Language – язык шейдеров OpenGL ES. Аббревиатура ES означает Embedded System (встраиваемые системы).

R

RGBA Формат представления цвета: R (red – красный), G (green – зеленый), B (blue – синий) и A (alpha – альфа-канал).

U

uniform-переменная Переменная, используемая для передачи данных в вершинный или фрагментный шейдер.

V

varying-переменная Переменная, используемая для передачи данных из вершинного во фрагментный шейдер.

A

альфа-смешивание Процесс использования альфа-канала («A») в RGBA для смешивания цветов двух или более объектов.

альфа-канал Значение, определяющее степень прозрачности (0.0 – полностью прозрачный, 1.0 – полностью непрозрачный) объекта. Это значение используется при альфа-смешивании.

Б

ближняя плоскость отсечения Плоскость, обозначающая самую ближнюю от точки наблюдения границу видимого объема.

буфер Блок памяти, выделенный и предназначенный для хранения данных особого вида, таких как значения цвета или глубины.

буфер глубины Область памяти, используемая механизмом удаления скрытых поверхностей. Хранит значения глубины (z-значения) всех фрагментов.

буфер цвета Область памяти, в которой система WebGL рисует объекты. По окончании рисования, содержимое этого буфера автоматически отображается на экране.

В

вершинный шейдер Программа шейдера, обрабатывающая информацию о вершинах.

вершины индекс Число, присвоенное элементу информации о вершине, хранящемуся в буферном объекте. Отсчет начинается с 0 и увеличивается с шагом 1.

видимый объем Область пространства, отображаемая на экране. Объекты за пределами видимого объема не отображаются.

Г

глубина (значение) Z-значение фрагмента относительно точки наблюдения.

Д

дальняя плоскость отсечения Плоскость, обозначающая самую дальнюю от точки наблюдения границу видимого объема.

З

затенение Процесс применения затенения ко всем поверхностям объекта.

И

изображение Прямоугольный массив пикселей.

изображение текстуры Изображение, используемое для наложения текстуры. Часто называют просто текстурой.

индекс (вершины) См. термин *вершины индекс*.

К

контекст Объект JavaScript, реализующий методы для рисования в элементе `<canvas>`.

координаты текстуры 2-мерные координаты, используемые для доступа к изображению текстуры.

Л

локальные координаты Координаты вершины в локальной системе координат (в системе координат данного конкретного объекта). (См. также термин *мировые координаты*.)

М

матрица вида Матрица для преобразования координат вершин, видимых с заданной точки наблюдения.

матрица вида проекции Матрица, получаемая умножением матрицы проекции на матрицу вида.

матрица модели Матрица, используемая для перемещения, вращения или масштабирования объекта.

матрица модели вида Матрица, получаемая умножением матрицы вида на матрицу модели.

матрица модели вида проекции Матрица, получаемая умножением матрицы проекции на матрицу модели вида.

матрица ортогональной проекции Матрица, используемая для определения видимого объема в форме параллелепипеда – левой, правой, нижней, верхней, ближней и дальней плоскостей отсечения. Объекты, находящиеся ближе к дальней плоскости отсечения, не уменьшаются в размерах.

матрица перспективной проекции Матрица, используемая для определения видимого объема в форме четырехгранной пирамиды. Объекты, находящиеся ближе к дальней плоскости отсечения, уменьшаются в соответствии с перспективой.

матрица проекции Обобщенный термин для обозначения матрицы ортогональной или перспективной проекции.

мировые координаты Координаты полученные умножением матрицы модели на локальные координаты вершины трехмерной модели.

Н

направленное освещение Освещение, когда лучи света от источника распространяются параллельно друг другу.

наложение текстуры Процесс применения (наложения) изображения текстуры к поверхности объекта.

наложение тени Процесс определения и рисования теней, отбрасываемых объектами.

нормаль Воображаемая линия, перпендикулярная к поверхности многоугольника и представленная вектором типа `vec3`. Также называется вектором нормали.

О

объект буфера Объект WebGL, используемый для хранения массива элементов с информацией о вершинах.

объект буфера кадра Объект WebGL, используемый для рисования в памяти.

объект программы Объект WebGL, предназначенный для управления объектами шейдеров.

объект текстуры Объект WebGL, предназначенный для управления изображением текстуры.

объект шейдера Объект WebGL, предназначенный для управления шейдером.

отсечение Операция, определяющая область внутри трехмерной сцены, которая будет нарисована. Все, что не попадает в эту область, отображаться не будет.

П

переменная-атрибут Переменная, используемая для передачи данных в вершинный шейдер.

пиксель Элемент рисунка. Имеет значение RGBA или RGB.

подключение Процесс установления связи между двумя существующими объектами. Сравните с термином *связывание*.

порядок расположения по столбцам Соглашение, описывающее способ хранения матрицы в массиве. В порядке расположения по столбцам, матрица хранится в массиве в виде последовательности столбцов.

преобразование Процесс перевода координат вершины в новые координаты в результате применения преобразования (трансляции (перемещения), масштабирования и т. д.).

Р

растеризация Процесс преобразования фигур из векторного формата представления во фрагментный (в виде массива пикселей или точек) для отображения на экране монитора.

С

связывание Процесс создания нового объекта с последующим установлением связи между этим объектом и контекстом отображения. Сравните с термином *подключение*.

семплер (sampler) Тип данных, используемый для доступа к изображению текстуры внутри фрагментного шейдера.

система координат вида Система координат с началом в точке наблюдения, взгляд направлен в сторону отрицательных значений оси Z, направление вверх совпадает с положительным направлением оси Y.

состояние готовности Используется при упоминании буфера кадра и указывает на готовность буфера к использованию для рисования.

Т

тексель (texel) Простейший элемент (*texture element* – элемент текстуры) изображения текстуры. Имеет значение RGB или RGBA.

текстурный слот Механизм управления множеством объектов текстуры.

точечный источник света Источник света, испускающий лучи во всех направлениях из одной точки.

туман Эффект постепенного перехода цвета с увеличением расстояния от наблюдателя. Туман часто используется для усиления эффекта глубины.

У

удаление скрытых поверхностей Процесс определения и сокрытия поверхностей и их частей, которые невидимы с данной точки наблюдения.

Ф

фоновое освещение Непрямое освещение. Свет, падающий на объект со всех сторон с одинаковой интенсивностью.

фрагмент Пиксель, сгенерированный на этапе растеризации и имеющий цвет, значение глубины, координаты текстуры и другие характеристики.

фрагментный шейдер Программа шейдера, обрабатывающая информацию о фрагментах.

Ш

шейдер Программа, реализующая основные функции рисования в WebGL. WebGL поддерживает вершинные и фрагментные шейдеры.



СПИСОК ЛИТЕРАТУРЫ

1. Bowman, Doug A., Ernst Kruijff, Joseph J. LaViola Jr, and Ivan Poupyrev. «3D User Interfaces: Theory and Practice», Addison-Wesley Professional (July 26, 2004).
2. Dunn, Fletcher, and Ian Parberry. «3D Math Primer for Graphics and Game Development», 2nd Edition. A K Peters/CRC Press (November 2, 2011).
3. Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes. «Computer Graphics: Principles and Practice in C», 2nd Edition. Addison-Wesley Professional (August 4, 1995).
4. Munshi, Aaftab, Dan Ginsburg, and Dave Shreiner. «OpenGL ES 2.0 Programming Guide», Addison-Wesley Professional (July 24, 2008).
5. Shreiner, Dave. The Khronos OpenGL ARB Working Group. «OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1», 7th Edition. Addison-Wesley Professional (July 21, 2009).¹

¹ На русском языке имеется только значительно более старое издание: Д. Шрайнер, М. Ву, Дж. Нейдер, Т. Девис «OpenGL. Руководство по программированию. Библиотека программиста. 4-е изд.», Питер, 2006, ISBN: 5-94723-827-6. – *Прим. перев.*

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Символы

- . (точка), оператор 213
- ; (точка с запятой), в GLSL ES 205
- [] (индексирование массивов), оператор 214
- <body>, элемент 36
- <canvas>, элемент 34
 - DrawingTriangle.html 34
 - DrawRectangle.js 36
 - HelloCanvas.html 41
 - HelloCanvas.js 41
 - координаты центра 47
 - отображение в систему координат WebGL 60
- очистка области рисования 40
- поддержка в браузерах 35
- получение ссылки 37, 42
- система координат 39
- установка цвета очистки 44
- 2-мерная графика
 - вершины разного цвета 165
 - сборка и растеризация геометрических фигур 165
 - восстановление отсеченных частей 262
 - вращение 244
 - вызов фрагментного шейдера 169
 - действие varying-переменных и процесс интерполяции 171
- трехмерная графика
 - альфа-смешивание 386
 - добавление фоновое освещения 315
 - загрузка 417
 - затенение 299
 - инструменты трехмерного моделирования 466
 - инструменты моделирования 417
 - освещение 299
 - источники света 300

- отраженный свет 302
- приложения
 - использование возможностей браузеров 29
 - простота публикации 29
 - создание в текстовом редакторе 28
- точечные источники света 322
- трехмерные модели
 - OBJViewer.js 421, 426
 - пользовательские объекты 424
 - формат файлов MTL 420
 - формат файлов OBJ 417, 419

А

- альфа-смешивание 383
- BlendedCube.js 387
- LookAtBlenderTriangles.js 384
- при наличии прозрачных и непрозрачных объектов 388
- реализация 383
- анимация 140
- RotatingTriangle.js 142
 - draw(), функция 145
 - вызов функции рисования 144
 - изменение угла поворота 148
- анонимные функции 74
- асинхронная загрузка изображений текстур 182
- аффинные преобразования 111
- бесконечность в однородных координатах 57

Б

- библиотеки, преобразований 131
- cuon-matrix.js 132
- RotatedTranslatedTriangle.js 137
- RotatedTriangle_Matrix4.js 133
- браузеры

- доступ к консоли 37
- доступ к локальным файлам
 - изображений 175
- настройки WebGL 470
- поддержка элемента `<canvas>` 35
- буфер глубины 280
- буферные объекты
 - запись координат вершин, индексов и цветов 291
 - создание нескольких объектов 155
- буферный объект
 - запись данных 96
 - описание 93
 - определение 90
 - присваивание переменной-атрибуту 99
 - разрешение присваивания переменной-атрибуту 101
 - создание 94
 - указание типа 95
- буферы 45
 - буфер глубины 45
 - буфер трафарета 45
 - буферы цвета 45
 - разновидности 45
 - цвета
 - перестановка 438
 - рисование 438

В

- веб-браузеры
 - доступ к консоли 37
 - использование возможностей в приложениях 29
 - поддержка элемента `<canvas>` 35
- векторные функции 70
- векторы
 - в GLSL ES 210
 - доступ к членам 213
 - конструкторы 212
 - операторы 216
 - присваивание значений 212
- вершинные шейдеры 49, 55, 243, 259
- передача данных 152
- примеры 203
- рисование точек 55
- вершины 50
 - рисование множества вершин 88
- вершины разного цвета 165
- вида, матрица 239, 242

- видимый диапазон 252
- видимый объем 252, 265
- точка наблюдения 252
- видимый объем
 - видимый диапазон 252, 265
 - в форме параллелепипеда
 - OrthoView.html 256
 - OrthoView.js 257
 - определение 252, 253
 - в форме четырехгранной пирамиды
 - определение 265, 267
- восстановление отсеченных частей 262
- вращение 115
 - матрицы преобразований 121, 124
- объектов
 - RotateObject.js 361
 - мышью 360
 - реализация 361
 - положительное 116
 - фигур 115
- вращения матрица
 - создание 121
- выбор
 - граней объектов 368
 - PickFace.js 368
 - объектов 363
 - PickObject.js 365
 - реализация 364
- вычисление
 - цвета для каждого фрагмента 326
 - цвета отраженного света 304

Г

- геометрические функции 445
- границы объектов
 - выбор 368
 - PickFace.js 368

Д

- данные, передача
 - MultiAttributeSize.js 154
 - в вершинные шейдеры 152
 - создание нескольких буферных объектов 155
 - с помощью `varying`-переменных 160
- директивы препроцессора в GLSL ES 233
- диффузное отражение 302, 303
- добавление
 - цвета для каждой грани куба 293

доступ
к консоли 37
доступ к членам
векторов и матриц 213

Е

единичная матрица 135

З

загрузка
трехмерных объектов 417
JavaScript 36
шейдеров из файлов 464
заднего плана, объекты 277
DepthBuffer.js 282
Z-конфликт 283
удаление скрытых поверхностей 280
запись
данных в буферный объект 96
зарезервированные слова в GLSL ES 206
затенение
трехмерных объектов 299
диффузное отражение 304
добавление фоновое освещения 315

И

идентификаторы, присваивание 35
иерархическая структура 331
изменение
значения far 260
значения near 260
положения точки наблюдения с
клавиатуры 249
цвета с помощью varying-переменных 160
элементов из JavaScript 258
изображения, наложение на
прямоугольник 174
выбор текстурного слота 184
извлечение цвета текселя 193
координаты текстуры 176, 179
наложение нескольких текстур 196
настройка параметров объекта
текстуры 187
передача координат текстуры
из вершинного шейдера во
фрагментный 193
передача текстурного слота
фрагментному шейдеру 192
поворот оси Y 183

подготовка текстур к использованию 183
присваивание изображения 190
указание типа объекта текстуры 185
индексы 291
интерполяция 171
источники света 300

К

карта теней 408
квалификаторы класса хранения
в GLSL ES 228
квалификаторы точности
в GLSL ES 231
клавиатура
изменение положения точки
наблюдения 249
ключевые слова в GLSL ES 206
комментарии, в GLSL ES 205
компиляция объектов шейдеров 351
компоновка объектов программ 355
консоль, доступ 37
константные выражения 215
конструкторы в GLSL ES 211
контекст
получение 38
координаты
локальная система координат 466
мировая система координат 467
однородные координаты 57
преобразования и системы координат 469
система координат WebGL 60
щелчка мышью 76
круглые точки
RoundedPoint.js 382
кубоиды 309
кубы 309

Л

листинги
BlendedCube.js 387
ClickedPoints.js 72
ColoredCube.js 295
ColoredPoints.js 81
ColoredTriangle.js 173
CoordinateSystem.js 451
CoordinateSystem_viewVolume.js 455
createProgram() 358
cube.mtl 420
cube.obj 418

DepthBuffer.js 282
 DrawingTriangle.html 34
 DrawRectangle.js 37
 FramebufferObject.js 398, 406
 HelloCanvas.html 41
 HelloCanvas.js 41
 HelloCube.js 289
 HelloPoint1.html 48
 HelloPoint1.js 48
 HelloPoint2.js 64
 HelloTriangle.js 105, 165
 HUD.html 372
 initShaders() 357
 JointModel.js 333
 LightedCube_ambient.js 316
 LightedCube.js 310
 вершинный шейдер 311
 обработка в JavaScript 314
 LightedTranslatedRotatedCube.js 320
 loadShader() 358
 LoadShaderFromFiles 464
 LookAtBlenderTriangles.js 384
 LookAtRotatedTriangles.js 246
 LookAtRotatedTriangles_mvMatrix.js 248
 LookAtTriangles.js 240
 LookAtTrianglesWithKeys.js 250
 LookAtTrianglesWithKeys_ViewVolume.js 263
 MultiAttributeColor.js 162
 MultiAttributeSize_Interleaved.js 157
 MultiAttributeSize.js 154
 MultiJointModel.js 340
 MultiJointModel_segment.js 345
 MultiPoint.js 90
 MultiTexture.js 197
 OBJViewer.js 421, 426
 OrthoView.html 256
 OrthoView.js 257
 PerspectiveView.js 269
 матрица модели 269
 PerspectiveView_mvp.js 274
 PickFace.js 368
 PointLightedCube.js 324
 PointLightedCube_perFragment.js 327
 ProgramObject.js 390
 RotatedTranslatedTriangle.js 137
 RotatedTriangle.js 118
 RotatedTriangle_Matrix4.html 132
 RotatedTriangle_Matrix4.js 133
 RotateObject.js 361

RotatingTriangle_contextLost.js 434
 RotatingTriangle.js 142
 изменение угла поворота 148
 draw(), функция 145
 вызов функции рисования 144
 RoundedPoint.js 382
 Shadow_highp.js 415
 Shadow.js 409, 413
 TexturedQuad.js 177
 TranslatedTriangle.js 113
 Zfighting.js 284
 массив с разнотипной информацией о
 вершинах 156
 пример вершинного шейдера 203
 пример фрагментного шейдера 204
 логические значения в GLSL ES 206
 локальная система координат 466

М

массивы
 доступ к членам 220
 конструирование 220
 перемежение 156
 присваивание 220
 типизированные массивы 98
 масштабирование
 матрицы преобразований 128
 матрица вида 239, 242
 матрица вращения
 создание 121
 транспонированная обратная матрица 460
 матрица масштабирования
 транспонированная обратная матрица 460
 матрица модели
 PerspectiveView 272
 матрица ортогональной проекции 263
 матрица перспективной проекции 268
 матрица преобразования
 транспонированная обратная матрица 460
 матрица трансляции
 транспонированная обратная матрица 460
 матрицы
 в GLSL ES 210
 доступ к членам 213
 единичная матрица 135
 конструкторы 212
 матрица модели 137
 PerspectiveView 272
 матрица перспективной проекции

- назначение 271
- четырёхгранная пирамида 268
- матрица проекции 457
- операторы 216
- определение 121
- присваивание значений 211
- транспонированная обратная матрица 319, 460
- умножение 121, 137, 217
- матрицы вращения
 - определение 123
- матрицы преобразований 121, 124
 - вращение 121, 124
 - масштабирование 128
 - перемещение 123
- матричные функции 446, 447
- методы
 - объекта Matrix4 134
- мировая система координат 467
- множества вершин
 - рисование
 - использование буферного объекта 93
- множества точек
 - рисование
 - использование буферного объекта 93
- множественные преобразования 131, 135
 - cuon-matrix.js 132
 - RotatedTriangle_Matrix4.js 133
- множество вершин
 - рисование
 - gl.drawArrays(), функция 102
 - запись данных в буферный объект 96
 - присваивание буферных объектов
 - переменным-атрибутам 99
 - разрешение присваивания буферных объектов переменным-атрибутам 101
 - создание буферного объекта 94
 - указание типа буферного объекта 95
- рисование простых фигур 104
- множество вершин, рисование 88
- множество точек
 - рисование
 - gl.drawArrays(), функция 102
 - запись данных в буферный объект 96
 - присваивание буферных объектов
 - переменным-атрибутам 99
 - разрешение присваивания буферных объектов переменным-атрибутам 101
 - создание буферного объекта 94

- указание типа буферного объекта 95
- множество точек, рисование 88
- модели матрица 137
- модель преобразований 137
- модель с единственным сочленением 332
- модель со множеством сочленений 339
 - MultiJointModel.js 340
- мышь
 - вращение объектов 360
 - координаты щелчка 76
 - обработка щелчков 75
 - рисование точек 71
 - ClickedPoints.js 72

Н

- наложение изображений 174
 - выбор текстурного слота 184
 - извлечение цвета текселя 193
 - координаты текстуры 176, 179
 - наложение нескольких текстур 196
 - настройка параметров объекта текстуры 187
- передача координат текстуры
 - из вершинного шейдера во фрагментный 193
- передача текстурного слота
 - фрагментному шейдеру 192
- поворот оси Y 183
- подготовка текстур к использованию 183
- присваивание изображения 190
- указание типа объекта текстуры 185
- наложение нескольких текстур 196
- наложение текстур 174
 - выбор текстурного слота 184
 - извлечение цвета текселя 193
 - координаты текстуры 176, 179
 - наложение нескольких текстур 196
 - настройка параметров объекта текстуры 187
- передача координат текстуры
 - из вершинного шейдера во фрагментный 193
- передача текстурного слота
 - фрагментному шейдеру 192
- поворот оси Y 183
- подготовка к использованию 183
- присваивание изображения 190
- указание типа объекта текстуры 185
- направление вверх 239

- направление взгляда 237, 239
 - LookAtRotatedTriangles.js 246
 - LookAtRotatedTriangles_mvMatrix.js 248
- направление вверх 239
- определение 237
- точка наблюдения 239
- точка направления взгляда 239
- направленность системы координат 450
 - и матрица проекции 457
 - удаление скрытых поверхностей 453
 - усеченная система координат и видимый объем 454
- направленный свет 301
 - диффузное отражение 304
 - затенение 304

О

- область рисования
 - определение 35
 - отображение в систему координат WebGL 60
- обработка щелчков мышью 75
- обработчики событий
 - регистрация 73
- общие функции 443
- объединение
 - нескольких преобразований 135
- объекты
 - StringParser, объект 428
 - буфера кадра 395
 - создание 399
 - буфера отображения 395
 - создание 401
- вращение
 - RotateObject.js 361
 - мышью 360
 - реализация 361
- выбор 363
 - PickObject.js 365
 - реализация 364
- выбор грани 368
 - PickFace.js 368
- определяемые пользователем 424
- составные
 - draw(), функция 337
 - JointModel.js 333
 - MultiJointModel.js 340
 - MultiJointModel_segment.js 345
 - иерархическая структура 331

- модель с единственным сочленением 332
- модель со множеством сочленений 339
- рисование 329
- рисование сегментов 343
- управление 329
- объекты программ 66
- однородные координаты 57
- операторов приоритеты в GLSL ES 221
- операторы
 - в GLSL ES 209
 - для работы с векторами и матрицами 216
- ориентация поверхности 306
- нормаль 306
- ортогональная проекция 253
- освещение
 - трехмерных объектов 299
 - добавление фоновое освещения 315
 - источники света 300
 - отраженный свет 302
 - направленный свет 300
 - перемещаемых и вращаемых объектов 317
 - рассеянный свет 300
 - точечный свет 300
- отсеченные части, восстановление 262

П

- передача данных
 - MultiAttributeSize.js 154
 - в вершинные шейдеры 152
 - создание нескольких буферных объектов 155
 - с помощью varying-переменных 160
- переднего плана, объекты 277
 - DepthBuffer.js 282
 - Z-конфликт 283
 - удаление скрытых поверхностей 280
- переключение шейдеров 389
 - ProgramObject.js 390
 - реализация 390
- переменные
 - uniform-переменные 82
 - получение ссылки 83
 - присваивание значений 84
 - в GLSL ES 206
 - uniform-переменные 230
 - varying-переменные 230
 - глобальные 227
 - квалификаторы класса хранения 228

- квалификаторы точности 231
- локальные 227
- переменные-атрибуты 229
 - в вершинных шейдерах 56
- преобразование типов 208
- присваивание значений 208
- соглашениям об именовании 65
- переменные-атрибуты
 - объявление 64
 - описание 63
 - получение ссылки 65
 - присваивание буферных объектов 99
 - присваивание значений 66
 - разрешение присваивания буферных объектов 101
- перемещение 112
 - матрицы преобразований 123
 - фигур 112, 123
- перестановка буферов цвета 438
- перспективная проекция 253
- повернутый треугольник, вид с определенной точки наблюдения 245
- подготовка текстуры к использованию 183
- положительное вращение 116
- полосы Маха 412
- получение ссылки
 - на uniform-переменную 83
- порядок выполнения в GLSL ES 205
- потеря контекста 432
 - реализация обработки 433
- преобразований библиотеки 131
 - cuon-matrix.js 132
 - RotatedTranslatedTriangle.js 137
 - RotatedTriangle_Matrix4.js 133
- преобразования
 - определение 111
- преобразования и системы координат 469
- приведение типов в GLSL ES 209
- Привет, куб 285
 - ColoredCube.js 295
 - добавление цвета для каждой грани куба 293
 - рисование объектов с использованием индексов и координат вершин 287
- примеры программ
 - BlendedCube.js 387
 - ClickedPoints.js 72
 - ColoredCube.js 295
 - ColoredPoints.js 81
 - ColoredTriangle.js 173
 - CoordinateSystem.js 451
 - CoordinateSystem_viewVolume.js 455
 - cube.mtl 420
 - cube.obj 418
 - cuon-utils.js 43
 - DepthBuffer.js 282
 - DrawingTriangle 34
 - DrawRectangle.js 37
 - FramebufferObject.js 398, 406
 - HelloCanvas.html 41
 - HelloCanvas.js 41
 - HelloCube.js 289
 - HelloPoint1.html 48
 - HelloPoint1.js 48
 - HelloPoint2.js 64
 - HelloTriangle.js 105, 165
 - HUD.html 372
 - JointModel.js 333
 - LightedCube_ambient.js 316
 - LightedCube.js 310
 - вершинный шейдер 311
 - обработка в JavaScript 314
 - LightedTranslatedRotatedCube.js 320
 - LoadShaderFromFiles 464
 - LookAtBlenderTriangles.js 384
 - LookAtRotatedTriangles.js 246
 - LookAtRotatedTriangles_mvMatrix.js 248
 - LookAtTriangles.js 240
 - сравнение с RotatedTriangle_Matrix4.js 243
 - LookAtTrianglesWithKeys.js 250
 - LookAtTrianglesWithKeys_ViewVolume.js 263
 - MultiAttributeColor.js 162
 - MultiAttributeSize_Interleaved.js 157
 - MultiAttributeSize.js 154
 - MultiJointModel.js 340
 - MultiJointModel_segment.js 345
 - MultiPoint.js 90
 - MultiTexture.js 197
 - OBJViewer.js 421, 426
 - OrthoView.html 256
 - OrthoView.js 257
 - PerspectiveView.js 269
 - матрица модели 269
 - PerspectiveView_mvp.js 274
 - PickFace.js 368
 - PickObject.js 365
 - PointLightedCube.js 324

PointLightedCube_perFragment.js 327
 ProgramObject.js 390
 RotatedTranslatedTriangle.js 137
 RotatedTriangle.js 118
 RotatedTriangle_Matrix4.html 132
 RotatedTriangle_Matrix4.js 133
 сравнение с LookAtTriangles.js 243
 RotatedTriangle_Matrix.js 125
 RotateObject.js 361
 RotatingTriangle_contextLost.js 434
 RotatingTriangle.js 142
 draw(), функция 145
 вызов функции рисования 144
 изменение угла поворота 148
 RoundedPoint.js 382
 Shadow_highp.js 415
 Shadow.js 409, 413
 TexturedQuad.js 177
 TranslatedTriangle.js 113
 Zfighting.js 284
 приоритеты операторов в GLSL ES 221
 присваивание
 буферных объектов переменным-атрибутам 99
 разрешение присваивания буферных объектов переменным-атрибутам 101
 присваивание значений
 векторы 212
 матрицы 211
 переменным
 в GLSL ES 208
 программ объекты
 компоновка 355
 создание 353
 удаление 354
 прямоугольники 37, 109
 рисование 37

Р

рабочий буфер цвета 438
 растеризация 152, 165, 166
 реализация
 альфа-смешивания 383
 BlendedCube.js 387
 LookAtBlenderTriangles.js 384
 вращения объектов 361
 выбора объектов 364
 PickObject.js 365

круглой точки
 RoundedPoint.js 382
 обработки потери контекста 433
 переключения шейдеров 390
 ProgramObject.js 390
 текстура 397
 теней 408
 Shadow_highp.js 415
 Shadow.js 409, 413
 увеличение точности 414
 эффекта индикации на лобовом стекле 371
 HUD.html 372
 регистрация обработчиков событий 73
 рисование
 точек
 по щелчку мышью 71
 рисование
 в буферах цвета 438
 множества точек/вершин 88
 gl.drawArrays(), функция 102
 запись данных в буферный объект 96
 использование буферного объекта 93
 присваивание буферных объектов
 переменным-атрибутам 99
 разрешение присваивания буферных объектов переменным-атрибутам 101
 указание типа буферного объекта 95
 объектов с использованием индексов и координат вершин 287
 прямоугольников 37, 109
 составных объектов 329
 точек
 ClickedPoints.js 72
 ColoredPoints.js 81
 gl.drawArrays(), функция 58
 gl.getAttribLocation(), функция 66
 gl.getUniformLocation(), функция 83
 gl.uniform4f(), функция 84
 gl.vertexAttrib3f(), функция 66
 HelloPoint2.js 64
 uniform-переменные 82
 uniform-переменные, получение
 ссылки 83
 uniform-переменные, присваивание значений 84
 вершинные шейдеры 55
 изменение цвета точки 79
 обработка щелчков мышью 75
 переменные-атрибуты 63

- переменные-атрибуты, присваивание значений 66
- получение ссылки на переменную-атрибут 65
- регистрация обработчиков событий 73
- система координат WebGL 60
- способ второй 62
- фрагментные шейдеры 58
- точки
 - способ первый 46
- треугольников 104
- фигур 104
 - анимация 140
 - библиотеки преобразований 131
 - вращение 115
 - из множества точек/вершин 88
 - масштабирование 128
 - матрицы перемещения 123
 - матрицы преобразований 121, 124
 - множественные преобразования 135
 - перемещение 112
- рисование сегментов, составные объекты 343

С

- сборка и растеризация геометрических фигур 165
- связывание
 - объекта буфера кадра 403
 - объекта буфера отображения 401
- семплы 221
- система координат
 - элемента <canvas> 39
- системы координат
 - и матрица проекции 457
 - локальная система координат 466
 - мировая система координат 467
 - направленность по умолчанию 450
 - преобразования и системы координат 469
 - удаление скрытых поверхностей 453
 - усеченная система координат и видимый объем 454
- смешивание (swizzling) 214
- соглашениям об именовании переменных 65
- соглашения по именованию методов WebGL 70
- соглашения по именованию в GLSL ES 206
- создание
 - объекта буфера кадра 399

- объекта буфера отображения 401
- объектов программ 353
- объектов шейдеров 350
- текстуры 400
- сочленения 331
 - JointModel.js 333
 - MultiJointModel.js 340
 - модель с единственным сочленением 332
 - модель со множеством сочленений 339
- спецификаторы класса хранения 65
- структуры
 - в GLSL ES 218
 - доступ к членам 219
 - конструирование 219
 - присваивание 219

Т

- тексели 174
 - форматы данных 191
- текстура
 - создание 400
- текстуры 174, 395
 - асинхронная загрузка изображений
 - текстур 182
 - подготовка к использованию 183
- текстуры, наложение 174
 - координаты текстуры 176, 179
 - подготовка к использованию 183
- тени 407
 - Shadow_highp.js 415
 - Shadow.js 409, 413
 - реализация 408
 - метод карты теней 408
 - увеличение точности 414
- типизированные массивы 98
- типизированные языки 56
- типы данных
 - в GLSL ES 206
 - векторы и матрицы 210
 - в языке GLSL ES 56
 - массивы 220
 - семплы 221
 - структуры 219
 - типизированные массивы 98
- точечные источники света 322
- точечный свет 301
- точка наблюдения 238, 239
 - LookAtTrianglesWithKeys.js 250
- видимый диапазон 252

- изменение положения с клавиатуры 249
- точка направления взгляда 239
- транспонированная обратная матрица 319, 460
- треугольники
 - рисование 104
- треугольники 236
 - анимация 140
 - вершины разного цвета 165
 - вращение 115, 244
 - множественные преобразования 135
 - перемещение 112
- тригонометрические функции 441

У

- удаление
 - объектов программ 354
 - объектов текстур 180
 - объектов шейдеров 350
- удаление скрытых поверхностей 280, 453
 - DepthBuffer.js 282
 - Z-конфликт 283
- умножение
 - векторов и матриц 217
 - матриц 137
 - матрицы на вектор 121
- управление потоком выполнения
 - в GLSL ES 222
- управление составными объектами 329
- усеченная система координат и видимый объем 454

Ф

- файлы, загрузка шейдеров 464
- фигуры
 - анимация 140
 - библиотеки преобразований 131
 - cuon-matrix.js 132
 - восстановление отсеченных частей 262
 - вращение 115
 - масштабирование 128
 - матрицы перемещения 123
 - матрицы преобразований 121, 124
 - множественные преобразования 135
 - перемещение 112
 - рисование 104
 - HelloTriangle.js 105
 - рисование из множества вершин 88
 - список 107

- фоновое отражение 302, 303
- фоновый буфер цвета 438
- фрагментные шейдеры 50, 58
 - varying-переменные и процесс интерполяции 171
- вызов фрагментного шейдера 169
- извлечение цвета текселя 193
- передача текстурного слота 192
- примеры 204
- рисование точек 58
- сборка и растеризация геометрических фигур 165
- фрагменты 50, 58
- функции
 - abs(), функция 443
 - acos(), функция 442
 - all(), функция 447
 - any(), функция 447
 - asin(), функция 442
 - atan(), функция 442
 - cancelAnimationFrame(), функция 148
 - canvas.addEventListener(), функция 434
 - canvas.getContext(), функция 38
 - ceil(), функция 443
 - checkFace(), функция 370
 - clamp(), функция 444
 - click(), функция 75
 - cos(), функция 442
 - createProgram(), функция 358, 390
 - cross(), функция 445
 - ctx.fillRect(), функция 39
 - degrees(), функция 441
 - distance(), функция 445
 - document.getElementById(), функция 37, 42
 - dot(), функция 445
 - drawBox(), функция 343
 - drawSegment(), функция 345
 - draw(), функция
 - порядок операций 260
 - составные объекты 337
 - equal(), функция 447
 - exp2(), функция 443
 - exp(), функция 442
 - faceforward(), функция 446
 - floor(), функция 443
 - fract(), функция 443
 - getWebGLContext(), функция 43
 - gl.activeTexture(), функция 184

- gl.attachShader(), функция 354
- gl.bindBuffer(), функция 96
- gl.bindFramebuffer(), функция 403
- gl.bindRenderbuffer(), функция 402
- gl.bindTexture(), функция
 - 186, 402, 403, 405, 406, 407
- gl.blendFunc(), функция 385
- gl.bufferData(), функция 97
- gl.checkFramebufferStatus(), функция 405
- gl.clearColor(), функция 44
- gl.clear(), функция 45, 141
- gl.compileShader(), функция 351
- gl.createBuffer(), функция 94
- gl.createFramebuffer(), функция 400
- gl.createProgram(), функция 353
- gl.createRenderbuffer(), функция 401
- gl.createShader(), функция 350
- gl.createTexture(), функция 180
- gl.deleteBuffer(), функция 95
- gl.deleteFramebuffer(), функция 400
- gl.deleteProgram(), функция 354
- gl.deleteRenderbuffer(), функция 401
- gl.deleteShader(), функция 350
- gl.deleteTexture(), функция 180
- gl.depthMask(), функция 388
- gl.detachShader(), функция 354
- gl.disableVertexAttribArray(), функция 102
- gl.disable(), функция 281
- gl.drawArrays(), функция
 - 58, 92, 102, 106, 293
- gl.drawElements(), функция 287
- gl.enableVertexAttribArray(), функция 101
- gl.enable(), функция 280
- gl.framebufferRenderbuffer(), функция 404
- gl.framebufferTexture2D(), функция 403
- gl.getAttribLocation(), функция 66
- gl.getProgramInfoLog(), функция 356
- gl.getProgramParameters(), функция 355
- gl.getShaderInfoLog(), функция 352
- gl.getShaderParameter(), функция 352
- gl.getUniformLocation(), функция 83
- gl.linkProgram(), функция 355
- gl.pixelStorei(), функция 184
- gl.polygonOffset(), функция 284
- gl.readPixels(), функция 366
- gl.renderbufferStorage(), функция 402
- gl.shaderSource(), функция 351
- gl.texImage2D(), функция 190
- gl.texParameteri(), функция 187
- gl.translatef(), функция 132
- gl.uniform1f(), функция 86
- gl.uniform2f(), функция 86
- gl.uniform3f(), функция 86
- gl.uniform4f(), функция 84, 114
- gl.uniformMatrix4fv(), функция 127
- gl.useProgram(), функция 356
- gl.vertexAttrib1f(), функция 68
- gl.vertexAttrib2f(), функция 68
- gl.vertexAttrib3f(), функция 66, 68
- gl.vertexAttrib4f(), функция 68
- gl.vertexAttribPointer(), функция 100, 158
- gl.viewport(), функция 407
- greaterThanEqual(), функция 447
- greaterThan(), функция 447
- initShaders(), функция 54, 349, 357, 390
- initVertexBuffers(), функция
 - 92, 155, 166, 291
- inversesqrt(), функция 443
- length(), функция 445
- lessThanEqual(), функция 447
- lessThan(), функция 447
- loadShaderFile(), функция 465
- loadShader(), функция 358
- loadTexture(), функция 182
- log2(), функция 443
- log(), функция 443
- Matrix4.setOrtho(), функция 449
- Matrix4.setPerspective(), функция 449
- matrixCompMult(), функция 446
- max(), функция 444
- min(), функция 444
- mix(), функция 444
- mod(), функция 444
- normalize(), функция 446
- notEqual(), функция 447
- not(), функция 447
- popMatrix(), функция 343
- pow(), функция 442
- pushMatrix(), функция 343
- radians(), функция 441
- reflect(), функция 446
- refract(), функция 446
- requestAnimationFrame(), функция 145, 147
- setInterval(), функция 146
- setLookAt(), функция 239
- setOrtho(), функция 254

- setPerspective(), функция 267
- setRotate(), функция 133
- sign(), функция 443
- sin(), функция 441
- smoothstep(), функция 445
- sqrt(), функция 443
- step(), функция 445
- tan(), функция 442
- texture2DLod(), функция 448
- texture2DProjLod(), функция 448
- texture2DProj(), функция 448
- texture2D(), функция 194
- textureCubeLod(), функция 448
- textureCube(), функция 448
- vec4(), функция 57
- анонимные функции 74
- в GLSL ES 224
 - встроенные 227
 - квалификаторы параметров 226
 - прототипы 225
- геометрические 227, 445
- для работы с векторами 227
- для работы с матрицами 227
- для работы с текстурами 227
- для работы с угловыми величинами 227, 441
- матричные 446, 447
- обработчики событий
 - регистрация 73
- общего назначения 227
- общие 443
- тригонометрические 227, 441
- экспоненциальные 227, 442

Ц

цвет

- добавление для каждой грани куба 293
- изменение с помощью varying-переменных 160
- изменение цвета точки 79
- установка 38

цвет для каждого фрагмента, вычисление 326

Ч

- числовые значения в GLSL ES 206
- чувствительность к регистру в GLSL ES 205

Ш

- шейдеров объекты

- компиляция 351
- создание 350
 - initShaders(), функция 349
- удаление 350
- шейдеры
 - LoadShaderFromFiles 464
 - вершинные
 - пример 203
 - вершинные шейдеры 49, 55, 243, 259
 - передача данных 152
 - загрузка из файлов 464
 - инициализация 53
 - определение 31, 47
 - переключение 389
 - ProgramObject.js 390
 - реализация 390
 - структура программы с шейдерами 51
- фрагментные
 - пример 204
- фрагментные шейдеры 50, 58
 - varying-переменные и процесс интерполяции 171
- вызов фрагментного шейдера 169
- извлечение цвета текстуры 193
- передача текстурного слота 192
- рисование точек 58
- сборка и растеризация геометрических фигур 165
- язык шейдеров 31

Э

- экспоненциальные функции 442

элементы

- изменение из JavaScript 258

эффект индикации на лобовом стекле 371

- HUD.html 372
- реализация 371

А

- abs(), функция 443
- acos(), функция 442
- all(), функция 447
- any(), функция 447
- asin(), функция 442
- atan(), функция 442

В

- BlendedCube.js 387

Blender, инструмент трехмерного
моделирования 417, 466
bool, тип данных 208
break, инструкция в GLSL ES 223

C

cancelAnimationFrame(), функция 148
canvas.addEventListener(), функция 434
canvas.getContext(), функция 38
cdjgcdnf
 объекта Matrix4 134
ceil(), функция 443
checkFace(), функция 370
Chrome
 доступ к консоли 37
 доступ к локальным файлам
 изображений 175
 настройки WebGL 470
clamp(), функция 444
ClickedPoints.js 72
click(), функция 75
ColoredCube.js 295
ColoredPoints.js 81
ColoredTriangle.js 173
const, квалификатор класса хранения 228
continue, инструкция в GLSL ES 223
CoordinateSystem.js 451
CoordinateSystem_viewVolume.js 455
cos(), функция 442
createProgram(), функция 358, 390
cross(), функция 445
ctx.fillRect(), функция 39
cube.mtl 420
cube.obj 418
cuon-matrix.js 132
cuon-utils.js 43

D

degrees(), функция 441
DepthBuffer.js 282
discard, инструкция в GLSL ES 223
distance(), функция 445
document.getElementById(), функция 37, 42
dot(), функция 445
drawBox(), функция 343
DrawingTriangle.html 34
drawSegment(), функция 345
draw(), функция

порядок операций 260
составные объекты 337

E

equal(), функция 447
exp2(), функция 443
exp(), функция 442

F

faceforward(), функция 446
far, значение
 изменение 260
Firefox
 доступ к консоли 37
 доступ к локальным файлам
 изображений 175
 настройки WebGL 471
Float32Array, объект 98
float, тип данных 208
floor(), функция 443
for, инструкция в GLSL ES 223
fract(), функция 443
FramebufferObject.js 398, 406

G

getWebGLContext(), функция 43
gl.activeTexture(), функция 184
gl.attachShader(), функция 354
gl.bindBuffer(), функция 96
gl.bindFramebuffer(), функция 403
gl.bindRenderbuffer(), функция 402
gl.bindTexture(), функция
 186, 402, 403, 405, 406, 407
gl.blendFunc(), функция 385
gl.bufferData(), функция 97
gl.checkFramebufferStatus(), функция 405
gl.clearColor(), функция 44
gl.clear(), функция 45, 141
gl.compileShader(), функция 351
gl.createBuffer(), функция 94
gl.createFramebuffer(), функция 400
gl.createProgram(), функция 353
gl.createRenderbuffer(), функция 401
gl.createShader(), функция 350
gl.createTexture(), функция 180
gl.deleteBuffer(), функция 95
gl.deleteFramebuffer(), функция 400
gl.deleteProgram(), функция 354

- gl.deleteRenderbuffer(), функция 401
 - gl.deleteShader(), функция 350
 - gl.deleteTexture(), функция 180
 - gl.depthMask(), функция 388
 - gl.detachShader(), функция 354
 - gl.disableVertexAttribArray(), функция 102
 - gl.disable(), функция 281
 - gl.drawArrays(), функция 58, 92, 102, 106, 293
 - gl.drawElements(), функция 287
 - gl.enableVertexAttribArray(), функция 101
 - gl.enable(), функция 280
 - gl.framebufferRenderbuffer(), функция 404
 - gl.framebufferTexture2D(), функция 403
 - gl.getAttribLocation(), функция 66
 - gl.getProgramInfoLog(), функция 356
 - gl.getProgramParameters(), функция 355
 - gl.getShaderInfoLog(), функция 352
 - gl.getShaderParameter(), функция 352
 - gl.getUniformLocation(), функция 83
 - gl.linkProgram(), функция 355
 - gl.pixelStorei(), функция 184
 - gl.polygonOffset(), функция 284
 - gl.readPixels(), функция 366
 - gl.renderbufferStorage(), функция 402
 - gl.shaderSource(), функция 351
 - GLSL ES (OpenGL ES shading language)
 - break, инструкция 223
 - continue, инструкция 223
 - discard, инструкция 223
 - for, инструкция 223
 - if-else, инструкция 222
 - if, инструкция 222
 - векторы и матрицы 210
 - директивы препроцессора 233
 - комментарии 205
 - конструкторы 211
 - логические значения 206
 - обзор 204
 - операторы 209
 - приоритеты 221
 - переменные 206
 - uniform-переменные 230
 - varying-переменные 230
 - глобальные 227
 - квалификаторы класса хранения 228
 - квалификаторы точности 231
 - ключевые и зарезервированные слова 206
 - локальные 227
 - переменные-атрибуты 229
 - присваивание значений 208
 - соглашения по именованию 206
 - порядок выполнения 205
 - приведение типов 209
 - простые типы 207
 - типы данных 206, 207
 - массивы 220
 - семплы 221
 - структуры 219
 - (точка с запятой) 205
 - управление потоком выполнения 222
 - функции 224
 - встроенные 227
 - квалификаторы параметров 226
 - прототипы 225
 - числовые значения 206
 - чувствительность к регистру 205
 - GLSL ES (язык шейдеров Open GL 52
 - GLSL ES (язык шейдеров OpenGL)
 - типы данных 56
 - gl.texImage2D(), функция 190
 - gl.texParameteri(), функция 187
 - gl.Translatef(), функция 132
 - gl.uniform1f(), функция 86
 - gl.uniform2f(), функция 86
 - gl.uniform3f(), функция 86
 - gl.uniform4f(), функция 84, 114
 - gl.uniformMatrix4fv(), функция 127
 - gl.useProgram(), функция 356
 - gl.vertexAttrib1f(), функция 68
 - gl.vertexAttrib2f(), функция 68
 - gl.vertexAttrib3f(), функция 66, 68
 - gl.vertexAttrib4f(), функция 68
 - gl.vertexAttribPointer(), функция 100, 158
 - gl.viewport(), функция 407
 - greaterThanEqual(), функция 447
 - greaterThan(), функция 447
- ## Н
- HelloCube.js 289
 - HelloPoint1.html 48
 - HelloPoint1.js 48
 - HelloPoint2.js 64
 - HelloTriangle.js 105, 165
 - HTML5
 - <body>, элемент 36

`<canvas>`, элемент
 `DrawingTriangle.html` 34
 `DrawRectangle.js` 36
 `HelloCanvas.html` 41
 `HelloCanvas.js` 41
 координаты центра 47
 отображение в систему координат
 WebGL 60
 очистка области рисования 40
 поддержка в браузерах 35
 получение ссылки 37, 42
 система координат 39
 установка цвета очистки 44
 `DrawingTriangle.html` 34
 определение 27
HUD.html 372

I

`if-else`, инструкция в GLSL ES 222
`if`, инструкция в GLSL ES 222
`initShaders()`, функция 54, 349, 357, 390
`initVertexBuffers()`, функция 92, 155, 166, 291
`int`, тип данных 208
`inversesqrt()`, функция 443

J

JavaScript
 загрузка 36
 изменение содержимого
 HTML-элементов 258
 область рисования, отображение в
 систему координат WebGL 60
`JointModel.js` 333

K

`kbcnbyub^PickObject.js` 365
`kbcnbyub^RotatedTriangle_Matrix.js` 125
Khronos Group 31

L

`length()`, функция 445
`lessThanEqual()`, функция 447
`lessThan()`, функция 447
`LightedCube_ambient.js` 316
`LightedCube.js` 310
 вершинный шейдер 311
 обработка в JavaScript 314

`LightedTranslatedRotatedCube.js` 320
`loadShaderFile()`, функция 465
`LoadShaderFromFiles` 464
`loadShader()`, функция 358
`loadTexture()`, функция 182
`log2()`, функция 443
`log()`, функция 443
`LookAtBlenderTriangles.js` 384
`LookAtRotatedTriangles.js` 246
`LookAtRotatedTriangles_mvMatrix.js` 248
`LookAtTriangles.js` 240
 сравнение с `RotatedTriangle_Matrix4.js` 243
`LookAtTrianglesWithKeys.js` 250
`LookAtTrianglesWithKeys_ViewVolume.js` 263

M

`Matrix4.setOrtho()`, функция 449
`Matrix4.setPerspective()`, функция 449
`Matrix4`, объект, поддерживаемые свойства
 и методы 134
`matrixCompMult()`, функция 446
`max()`, функция 444
`min()`, функция 444
MIPMAP, формат текстур 189
`mix()`, функция 444
`mod()`, функция 444
MTL, формат файлов 420
`MultiAttributeColor.js` 162
`MultiAttributeSize_Interleaved.js` 157
`MultiAttributeSize.js` 154
`MultiJointModel.js` 340
`MultiJointModel_segment.js` 345
`MultiPoint.js` 90
`MultiTexture.js` 197

N

`near`, значение
 изменение 260
`normalize()`, функция 446
`notEqual()`, функция 447
`not()`, функция 447

O

`OBJViewer.js` 421, 426
OBJ, формат файлов 417, 419
OpenGL функции
 соглашения по именованиям 70
`OrthoView.html` 256

OrthoView.js 257

Р

PerspectiveView.js 269

матрица модели 269

PerspectiveView_mvp.js 274

PickFace.js 368

PickObject.js 365

PointLightedCube.js 324

PointLightedCube_perFragment.js 327

popMatrix(), функция 343

pow(), функция 442

ProgramObject.js 390

pushMatrix(), функция 343

R

radians(), функция 441

reflect(), функция 446

refract(), функция 446

requestAnimationFrame(), функция 145, 147

RGBA, формат 39

RGB, формат 39

RotatedTranslatedTriangle.js 137

RotatedTriangle.js 118

RotatedTriangle_Matrix4.html 132

RotatedTriangle_Matrix4.js 133

сравнение с LookAtTriangles.js 243

RotatedTriangle_Matrix.js 125

RotateObject.js 361

RotatingTriangle_contextLost.js 434

RotatingTriangle.js 142

draw(), функция 145

вызов функции рисования 144

изменение угла поворота 148

RoundedPoint.js 382

S

setInterval(), функция 146

setLookAt(), функция 239

setOrtho(), функция 254

setPerspective(), функция 267

setRotate(), функция 133

Shadow_highp.js 415

Shadow.js 409, 413

sign(), функция 443

sin(), функция 441

smoothstep(), функция 445

sqrt(), функция 443

step(), функция 445

StringParser, объект 428

T

tan(), функция 442

texture2DLod(), функция 448

texture2DProjLod(), функция 448

texture2DProj(), функция 448

texture2D(), функция 194

textureCubeLod(), функция 448

textureCube(), функция 448

TexturedQuad.js 177

TranslatedTriangle.js 113

U

uniform-переменные 82

получение ссылки 83

присваивание значений 84

V

varying-переменные

и процесс интерполяции 171

vec4(), функция 57

W

WebGL

достоинства 27

и OpenGL 30

история происхождения 30

настройка браузеров 470

определение 26

система координат 60

соглашения по именованию методов 70

структура приложений 31

структура программы 51

установка цвета 44

шейдеры 31

Z

Zfighting.js 284

Z-конфликты 283

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.**

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.aliants-kniga.ru.**

Оптовые закупки: тел. (499) 782-38-89

Электронный адрес: **books@aliants-kniga.ru.**

Коичи Мацуда, Роджер Ли

WebGL: программирование трехмерной графики

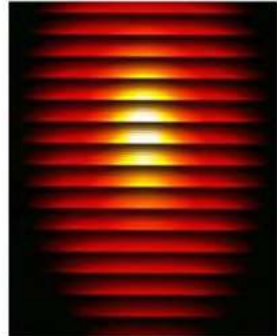
Главный редактор	<i>Мовчан Д. А.</i> dmkpress@gmail.com
Перевод с английского	<i>Киселев А. Н.</i>
Корректор	<i>Синяева Г. И.</i>
Верстка	<i>Паранская Н. В.</i>
Дизайн обложки	<i>Мовчан А. Г.</i>

Формат 70×100¹/₁₆. Гарнитура «Петербург».

Печать офсетная. Усл. печ. л. 40,14.

Тираж 200 экз.

Веб-сайт издательства: www.dmk.ru



WebGL:

программирование трехмерной графики

Владение технологией WebGL помогает создавать сложные интерактивные приложения с трехмерной графикой, выполняющиеся в веб-браузерах без поддержки каких-либо расширений. Данная книга поможет быстро освоить азы технологии WebGL, даже при отсутствии знаний HTML5, JavaScript, трехмерной графики, математики или OpenGL.

Вы будете учиться на реалистичных примерах, постепенно приобретая новые навыки по мере движения от простого к сложному, конструируя визуально привлекательные веб-страницы и веб-приложения с трехмерной графикой на основе WebGL. Пионеры в области мультимедиа, трехмерной графики и WebGL, **Коичи Мацуда** и **Роджер Ли**, предлагают вашему вниманию простое введение в ключевые аспекты WebGL, плюс 100 примеров программ, доступных для загрузки, каждая из которых иллюстрирует отдельный аспект WebGL.

Вы пройдете путь от простых приемов, таких как отображение, анимация и текстурирование треугольников, до продвинутых, таких как эффект тумана, затенение, переключение шейдеров и отображение трехмерных моделей, сгенерированных с помощью Blender или других подобных инструментов. Эта книга не только познакомит вас с наиболее эффективными приемами использования WebGL, но и даст вам библиотеку кода, на основе которой вы сможете начать разработку своих проектов.

Книга охватывает следующие темы:

- происхождение WebGL, базовые понятия, свойства, преимущества и интеграция с другими веб-стандартами;
- как с помощью `<canvas>` и основных функций WebGL организовать вывод трехмерной графики;
- разработка шейдеров на языке OpenGL ES Shading Language (GLSL ES);
- рисование трехмерных сцен: представление взгляда пользователя, управление видимым объемом, создание объектов и перспектива;
- увеличение реализма за счет освещения и применения иерархических объектов;
- продвинутые приемы: управление объектами, эффект индикации на лобовом стекле, альфа-смешивание, переключение шейдеров и многие другие;
- приложения в конце книги содержат ценную информацию по разным ключевым аспектам, начиная от систем координат до матриц, приемов загрузки шейдеров и настройки браузеров.

Интернет-магазин:
www.dmkpress.com

Книга - почтой:
orders@aliants-kniga.ru

Оптовая продажа:
"Альянс-книга"
Тел.: (499) 782-3889
books@aliants-kniga.ru


Addison-Wesley


ИЗДАТЕЛЬСТВО
www.dmk.pf

ISBN 978-5-97060-146-4



9 785970 601464 >