МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное
образовательное учреждение высшего образования
«ПЕРМСКИЙ ГОСУДАРСТВЕННЫЙ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ»

К. В. РЯБИНИН

ВЫЧИСЛИТЕЛЬНАЯ ГЕОМЕТРИЯ И АЛГОРИТМЫ КОМПЬЮТЕРНОЙ ГРАФИКИ

РАБОТА С 3D-ГРАФИКОЙ СРЕДСТВАМИ OPENGL

Допущено методическим советом Пермского государственного национального исследовательского университета в качестве учебного пособия для студентов, обучающихся по направлению подготовки бакалавров «Прикладная математика и информатика»

Пермь 2017

УДК 514+004.92
ББК 22.151+32.973–018.2
Р 98

Рябинин К. В.

Р 98 Вычислительная геометрия и алгоритмы компьютерной графики. Работа с 3D-графикой средствами OpenGL: учеб. пособие / К. В. Рябинин; Перм. гос. нац. исслед. ун-т. – Пермь, 2017. – 100 с.: ил.

ISBN 978-5-7944-2722-6

Пособие нацелено на изучение студентами основ вычислительной геометрии и компьютерной графики, приобретение практических навыков и компетенций разработки приложений, синтезирующих изображения в реальном времени с использованием аппаратного ускорения обработки графики на современных GPU. Излагаемый теоретический материал в равной степени относится ко всем низкоуровневым стандартам вывода графики; практическая часть приводится на примере открытого интерфейса OpenGL.

Предназначено для студентов, обучающихся по направлению подготовки бакалавров «Прикладная математика и информатика», изучающих дисциплину «Вычислительная геометрия и алгоритмы компьютерной графики», а также будет полезно студентам других направлений и специальностей, изучающим основы программного построения изображений средствами ЭВМ и осваивающим современные технологии разработки мультимедийных информационных систем.

Ил. 20. Библиогр. 12 назв.

УДК 514+004.92
ББК 22.151+32.973–018.2

Печатается по решению редакционно-издательского совета
Пермского государственного национального исследовательского университета

Рецензенты: к. т. н., доцент каф. информатики и вычислительной техники Перм. гос. гум.-пед. ун-та ***И. П. Половина***;
ГК «ИВС» (рецензенты – к. т. н. ***И. Ф. Федорищев***,
директор по инновациям ГК «ИВС» ***А. Н. Полещук***)

ISBN 978-5-7944-2722-6

Оглавление

Введение

Компьютерная графика как совокупность методов преобразования данных в визуальные образы с помощью ЭВМ занимает важное место в сфере современных информационных технологий. Она находит применение в индустрии развлечений (создание игр и специальных эффектов для фильмов), в науке и образовании (визуализация результатов научных экспериментов и создание наглядных пособий), в промышленности (автоматизированное проектирование и прототипирование), в средствах массовой информации (создание рекламных и информационных роликов), в бизнесе (визуализация бизнес-процессов и эконометрических данных) и даже в человеческом быту (графические интерфейсы пользователя к различным вычислительным устройствам).

Научной основой компьютерной графики является вычислительная геометрия – раздел дискретной математики, изучающий вопросы решения геометрических задач средствами ЭВМ.

Данное методическое пособие освещает основные вопросы разработки компьютерных программ для синтеза изображений в реальном времени. Рассматриваются математические и алгоритмические основы компьютерной графики, необходимые для управления процессом подготовки и отображения виртуальных графических сцен различной сложности. Изложенный материал включает в себя теоретические и практические аспекты программирования графики и предполагает реализацию описываемых приёмов по мере прочтения.

Алгоритмическая и элементная базы компьютерной графики развиваются с каждым годом: выходят всё новые версии видеокарт, драйверов, графических API; потентуются новые приёмы синтеза изображений. Однако существует некоторое алгоритмическое «ядро», сформированное ещё в 60–70-е г. XX в. и остающееся неизменным до сих пор. Данное пособие концентрируется в первую очередь именно на использовании этого «ядра».

По мере изложения периодически будут использоваться отсылки к «настоящему времени» (при описании тех или иных технологий, соответствия используемого программного обеспечения тем или иным стандартам и т. п.). В этом контексте под «настоящим временем» понимается *первая половина 2017 г.* Вполне вероятно, что несколько лет спустя ситуация кардинально изменится, однако учитывая историю развития компьютерной графики в последние полвека, алгоритмическое «ядро» будет актуально ещё долго – как минимум до тех пор, пока в историю не отойдут привычные нам сегодня средства отображения информации, такие как мониторы, дисплеи, проекторы, принтеры и типографские станки.

Следует сразу отметить, что данное пособие не является полноценным учебником или техническим руководством по компьютерной графике; скорее, оно представляет собой расширенное введение в эту область. Оно ориентировано на начинающих и ставит своей целью подготовить почву для изучения более серьёзных книг (таких как, например, «OpenGL Суперкнига» [1]), онлайн-курсов (например, «Neon-Helium» [2]) и прочих источников. Кроме того, оно включает в себя материал нескольких первых лекций курса «Вычислительная геометрия и алгоритмы компьютерной графики», который читается в Пермском государственном национальном исследовательском университете.

Предполагается, что читатель обладает базовыми знаниями в области геометрии и линейной алгебры, а также знаком с программированием на языках C/C++.

Работа с графикой рассматривается на примере стандарта OpenGL ввиду его мультиплатформенности. Однако все описанные методы и алгоритмы носят универсальный характер, поэтому, ознакомившись с данным пособием, читатель без труда сможет освоить и другие стандарты (например, Direct3D или Metal).

Для практической реализации описанных методов и алгоритмов читателю понадобится компьютер с поддержкой OpenGL версии не ниже 3.3, компилятор языка C++, а также набор библиотек, включающих реализацию функций стандарта OpenGL и вспомогательных операций.

Данное методическое пособие предполагает работу с графикой *на низком уровне*, т. е. на уровне, максимально приближенном к графическому оборудованию (видеокарте, GPU), чтобы у читателя сложилось глубокое понимание процессов синтеза изображений средствами современных ЭВМ.

1. Формирование изображений в компьютерной графике

Наиболее общий вид цепочки преобразований, приводящей к построению изображения средствами ЭВМ, представлен на рис. 1.

Рис. 1. Процесс визуализации

Сцена – это совокупность объектов, подлежащих отображению, описанная при помощи некоторой математической модели. Визуализация – это процесс преобразования математической модели сцены в вид, пригодный для показа на имеющихся устройствах вывода. Отображение – это процесс демонстрации итогового изображения человеку средствами имеющихся устройств вывода.

Как правило, отображение осуществляется посредством монитора и требует для этого подготовки *двумерного растрового изображения*, т. е. структуры данных в виде прямоугольной матрицы заданного размера, каждая ячейка которой содержит кодировку определённого цвета. Для кодирования чаще всего используется *цветовая модель RGB*, в которой каждый цвет представлен в виде яркости красного, зелёного и синего. Такая модель достаточно хорошо подходит для современных мониторов, поскольку они обычно представляют собой набор ячеек прямоугольной (как правило – квадратной) формы, каждая из которых может излучать свет красного, зелёного и синего цветов с контролируемой интенсивностью.

Методы и средства компьютерной графики могут быть классифицированы по самым разным критериям, но чаще всего разделение проводится по размерности пространства, с элементами которого ведётся работа, и по типу математической модели.

По размерности пространства принято выделять одномерную (1D), двумерную (2D), трёхмерную (3D) графику и графику высших порядков. По типу математической модели выделяют растровую и векторную графику. Растровая графика предполагает представление сцены в виде сетки элементов одинаковой формы. Векторная графика предполагает математическое описание элементов сцены.

Современное аппаратное обеспечение (GPU) нацелено на работу с 3D-графикой, так как она является наиболее естественной для человека и выступает, по сути, математическим обобщением не менее привычной 2D-графики. Однако большая часть современных средств отображения способна воспроизводить только плоские (2D) изображения, т. е. встаёт задача понижения размерности пространства в процессе визуализации.

С точки зрения представления сцены на GPU используется векторная модель, потому что в трёхмерном случае она оказывается значительно более эффективной по времени и памяти, чем растровая. Однако, как уже отмечалось выше, для отображения на экране необходимо растровое изображение, т. е. в процессе визуализации встаёт задача преобразования векторной сцены в растр.

Таким образом, если речь идёт о работе с 3D-графикой на современных GPU и отображении результата на мониторе, математическая модель сцены является векторной, а этап визуализации включает в себя проекцию сцены в двумерное пространство, растеризацию и вычисление цвета каждой точки растра в соответствии с параметрами объектов сцены. Визуализацию иначе ещё называют *рендерингом* (от англ. *Rendering* – преобразование одного в другое).

Представление векторной математической модели сцены на сегодняшний день является стандартизированным и единообразным для подавляющего большинства современных GPU. Все объекты сцены представляются в виде множества многоугольников, аппроксимирующих их поверхности. Многоугольники представляются

вершинами и связями между ними. Такую векторную модель иначе ещё называют *полигональной* (от гр. *полигон* – многоугольник).

Подход на основе полигонализации применяется в силу наличия эффективных алгоритмов для растеризации и вычисления визуальных свойств поверхностей. Обработка поверхностей как дискретных сеток, в свою очередь, хорошо ложится на логическую природу ЭВМ как конечного автомата.

Более того, большая часть графического оборудования предназначена для работы с сетками треугольников, а не произвольных многоугольников, так как вершины треугольника гарантированно лежат в одной плоскости. Это позволяет использовать простые и эффективные алгоритмы интерполяции параметров поверхности, дискретно заданных в узлах аппроксимирующей её сетки.

2. Уровни работы с графикой

Любое достаточно сложное приложение, работающее с компьютерной графикой, «в разрезе» выглядит так, как показано на рис. 2.

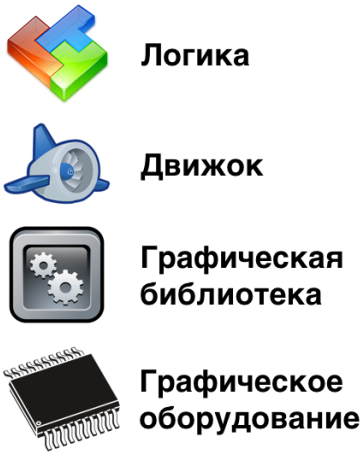

Рис. 2. Структура графического приложения

Основной вопрос, на который должен ответить разработчик перед началом создания графического приложения: с какого уровня начинать писать собственный программный код?

Программированием на уровне графического оборудования, как правило, занимаются лишь его производители.

Графическая библиотека, реализующая определённый стандарт абстрагирования от оборудования, напрямую взаимодействует с драйвером. Стандартами абстрагирования являются, например, OpenGL (англ. *Open Graphics Library* – открытая графическая библиотека; для настольных компьютеров под управлением различных ОС), OpenGLES (англ. *Open Graphics Library for Embedded Systems* – открытая графическая библиотека для встраиваемых

систем; для мобильных устройств под управлением различных ОС), WebGL (англ. *Web Graphics Library* – графическая библиотека для веб-приложений), Direct3D (для различных ЭВМ под управлением Windows и Windows Phone), Metal (для мобильных устройств под управлением iOS).

Реализацией стандартов в виде библиотек функций занимаются, как правило, производители оборудования или ОС. Графические библиотеки по природе своей являются универсальными и предполагают лишь минимальное повышение уровня API графического драйвера.

Графический движок (модуль графического расширения) призван повысить уровень API графической библиотеки для удобства реализации логики приложения. На уровне графической библиотеки работа производится с «сырыми» данными объектов сцены, такими как «вершина», «треугольник», «цвет» и т. д., лишёнными какой-либо семантики. На уровне движка выделяются сущности, в той или иной степени привязанные к предметной области приложения. Например, игровые движки могут работать в терминологии определённого класса игр, имея в своём API такие сущности, как «игрок», «противник», «оружие», «ландшафт», «препятствие», «индикатор очков» и пр. Существуют, однако, и универсальные движки, минимально повышающие уровень API графической библиотеки и работающие с абстрактными понятиями, такими, как «сцена», «трёхмерный объект», «источник света» и пр.

На сегодняшний день создано очень большое количество графических движков разной степени универсальности (и многие из них продолжают активно развиваться). По сути, существующее многообразие движков полностью покрывает потребности программистов, собирающихся создать очередное графическое приложение. Несмотря на это, иногда принимается решение разрабатывать собственный движок. Мотивация здесь может быть различна: желание иметь полный контроль над кодовой базой и независимость от сторонних разработчиков, желание набраться опыта, очень специфические условия задачи, проблемы юридического характера с лицензионными соглашениями и т. п. В общем случае «изобретение велосипеда» – скорее

плохо, чем хорошо, поэтому решение создавать собственный движок при работе над реальным проектом должно быть чётко обосновано.

Однако для начинающего разработчика создать собственный (вероятно, достаточно легковесный) движок – это хороший способ глубоко разобраться в вопросах компьютерной графики, а также набраться опыта в проектировании и реализации сложных программных систем.

Данное методическое пособие обходит вопросы выбора готового или реализации собственного движка стороной, концентрируя внимание на вопросах работы низкоуровневых графических библиотек. Обладая такими знаниями, читатель сможет как быстро освоить любой готовый движок, так и, при необходимости, создать свой собственный.

3. Графический конвейер

Как уже отмечалось выше, общепринятой математической моделью в 3D-графике выступает аппроксимация поверхностей объектов при помощи сеток из треугольников. Следует отметить, что методы компьютерной графики не ограничиваются таким подходом: существуют модели на основе гладких поверхностей, модели на основе элементов объёма (вокселей, от англ. *Voxel* – Volume Element, элемент объёма) и др. Однако полигональная модель на сегодняшний день является самой популярной и имеет аппаратную поддержку.

Атомарной управляемой геометрической единицей в полигональной модели является вершина. С ней может быть связан произвольный *набор атрибутов*, интерпретация которых осуществляется программистом. Чаще всего в этот набор входят, как минимум, пространственные координаты, но также может содержаться и дополнительная информация, например цвет или данные для его вычисления, информация о кривизне поверхности, скорость и направление перемещения вершины в пространстве и т. д.

Вершины объединяются в примитивы – минимальные отображаемые части объектов. Как правило, графическими библиотеками поддерживаются точки (отображение вершин в виде квадратов или кругов заданного размера), линии (отображение связей между вершинами в виде линий заданной толщины) и треугольники. Следует отметить, что графическое оборудование на низком уровне поддерживает только отображение треугольников. Точки и линии являются результатом автоматической триангуляции (покрытия множеством треугольников), производимой с использованием внутренних функций графической библиотеки. В связи с этим, при прочих равных, отображение примитивов-треугольников происходит быстрее, чем отображение примитивов-точек и примитивов-линий (графическая библиотека передаёт данные графическому драйверу без предварительных преобразований).

Данные о вершинах (их атрибуты), связях между ними и типах примитивов, в которые они входят, отправляются из основной про-

граммы (из оперативной памяти) в видеопамять (память видеокарты, к которой GPU имеет достаточно быстрый доступ). Как правило, отправка данных в видеопамять – это подготовительный этап, «загрузка сцены». Обычно он производится однократно, на фоне некоторого экрана загрузки, поскольку, как любая операция копирования, занимает относительно много времени.

Данные, находящиеся в видеопамяти, могут быть отправлены на т.н. *графический конвейер* (англ. *Pipeline*) – цепочку преобразований, в конце которой получается итоговое растровое изображение. Этапы работы графического конвейера приведены на рис. 3.

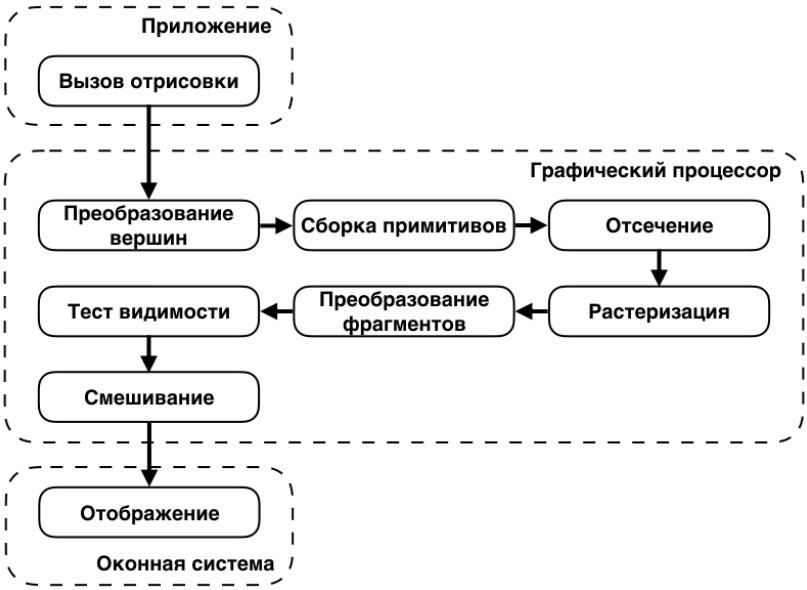

Рис. 3. Графический конвейер

Во время работы конвейера происходит следующее:

1. Вызов отрисовки (англ. *Draw Call*): отправка данных на графический конвейер.
2. Преобразование вершин (англ. *Vertex Transformation*): вычисление финальных координат вершин в пространстве по их ат-

рибутам (т.е. определение финального положения объектов на сцене), а также, при необходимости, динамическое порождение новых вершин (*тесселяция* поверхности).
3. Сборка примитивов (англ. *Primitive Assembling*): объединение отдельных вершин в примитивы.
4. Отсечение (англ. *Clipping*): отброс невидимых частей сцены.
5. Растеризация (англ. *Rasterization*): подбор точек растра (т. н. фрагментов) итогового изображения, покрывающих проекцию на экран каждого из примитивов.
6. Преобразование фрагментов (англ. *Fragment Transformation*): вычисление цвета (и других характеристик) каждой точки растра в соответствии с параметрами объекта, которому эта точка принадлежит.
7. Тест видимости (англ. *Visibility Test*): отброс невидимых фрагментов.
8. Смешивание (англ. *Blending*): объединение вычисленного цвета каждой точки растра с цветами, вычисленными для этих точек растра ранее.
9. Отображение (англ. *Displaying*): показ полученного изображения на экране.

4. Размещение объектов на сцене и проекция на экран

Для того чтобы создать итоговое изображение, примитивы, из которых состоят объекты сцены, должны быть спроецированы из пространства сцены на плоскость экрана. Однако перед этим, как правило, объекты должны быть размещены на сцене. Их положение в пространстве характеризуется положением принадлежащих им вершин. Согласованное изменение положения всех вершин объекта приведёт к изменению его положения в пространстве.

В трёхмерном пространстве у каждого объекта есть 6 степеней свободы: перемещение в трёх направлениях и три поворота вокруг взаимно перпендикулярных осей. В компьютерной графике к операциям размещения относят также масштабирование (изменение размера) и наклон объекта.

4.1. Аффинные преобразования

Как правило, каждый объект имеет некоторую свою *локальную* систему координат, в которой заданы его вершины. Например, если объект был создан в некотором 3D-редакторе, его вершины заданы в системе координат, определённой этим редактором. Если вместе с этим объектом на сцену будет загружен другой объект, созданный в том же редакторе, но отдельно от первого, скорее всего, эти два объекта пересекутся в пространстве, так как их вершины будут распределены в пересекающихся областях пространства.

Для того чтобы корректно расставить объекты на сцене, им необходимо задать некоторое положение в *глобальной* системе координат сцены.

Для перехода из одной системы координат в другую удобно использовать *аффинные преобразования* – отображения вида $f : \mathbb{R}^n \to \mathbb{R}^n$, сохраняющие свойство параллельности прямых линий.

Иначе говоря, аффинными называются преобразования, которые можно получить следующим образом:

1. Выбрать «новый» базис пространства с «новым» началом координат t.
2. Каждой точке v пространства поставить в соответствие точку $f(v)$, имеющую те же координаты относительно «новой» системы координат, что и v относительно «старой».

К аффинным преобразованиям относятся такие действия, как

1. Параллельный перенос.
2. Масштабирование.
3. Поворот.
4. Наклон.

Осуществлять их удобнее всего при помощи *матриц преобразования*. Любое аффинное преобразование может быть выражено в виде

$$f(v) = Mv + t,$$

где v – вектор-столбец координат исходной точки,
M – матрица преобразования,
t – вектор-столбец координат начала «новой» системы, выраженных в «старой» системе.

Для трёхмерного случая матрица M имеет размерность 3×3, а векторы v и t, соответственно, размерность 3. Однако для упрощения вида преобразований в компьютерной графике принято «вносить» вектор t в матрицу M. В этом случае, чтобы удовлетворить требованиям к операндам при умножении, её размерность увеличивают до 4×4, а размерность вектора v увеличивают до 4. Для этого M дополняется строкой $(0, 0, 0, 1)$, а вектор-столбец v – числом 1. Так происходит переход к *однородным координатам*. Аффинное преобразование в этом случае принимает вид

$$f(v) = Mv.$$

Матрица аффинного преобразования, по сути, описывает «новую» систему координат, к которой осуществляется переход при помощи соответствующего отображения (рис. 4).

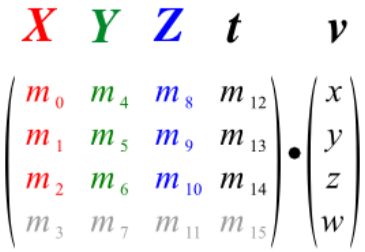

Рис. 4. Матрица аффинного преобразования

Векторы (m_0, m_1, m_2), (m_4, m_5, m_6) и (m_8, m_9, m_{10}), составленные из соответствующих элементов матрицы M, представляют собой разложение базиса «новой» системы координат в базисе «старой»; вектор (m_{12}, m_{13}, m_{14}) представляет собой начало «новой» системы относительно «старой».

Основываясь на таком принципе построения, нетрудно вывести матрицы для описанных выше аффинных преобразований:

$$T = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

где T – матрица переноса,
x, y, z – значения переноса по соответствующим осям.

$$S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

где S – матрица масштабирования,
s_x, s_y, s_z – значения масштаба по соответствующим осям.

$$R = \begin{pmatrix} x^2(1-c)+c & xy(1-c)-zs & xz(1-c)+ys & 0 \\ yx(1-c)+zs & y^2(1-c)+c & yz(1-c)-xs & 0 \\ xz(1-c)-ys & yz(1-c)+xs & z^2(1-c)+c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

где R – матрица поворота вокруг начала координат,
$c = \cos\theta$,
$s = \sin\theta$,
θ – угол поворота,
x, y, z – вектор, задающий ось поворота, причём $|(x, y, z)| = 1$.

Задание: Вывести матрицу наклона на угол θ.
Подсказка: Наклон рассмотреть отдельно в каждой координатной плоскости (в результате получится 6 различных матриц, соответствующих различным координатным плоскостям).

Суперпозиция аффинных преобразований достигается путём перемножения соответствующих матриц. Так, например, если необходимо изменить размер объекта и переместить его, каждую из его вершин необходимо умножить (слева) на матрицу

$$M = TS,$$

где T – матрица переноса,
S – матрица масштабирования.

Если требуется повернуть объект вокруг точки с координатами (x, y, z), каждую из его вершин необходимо умножить (слева) на матрицу

$$M = T_1 R T_2,$$

где T_1 – матрица переноса на (x, y, z),
R – матрица поворота,
T_2 – матрица переноса на $(-x, -y, -z)$.

При комбинировании матричных преобразований необходимо иметь в виду, что умножение матриц ассоциативно, но некоммутативно. Преобразование, записанное как произведение матриц, читается слева направо и представляет собой цепочку последовательных изменений системы координат объекта. Для того, чтобы правильно его интерпретировать, можно воспользоваться следующим неформальным правилом: воспринимать каждую матрицу в цепочке как

очередную команду объекту совершить то или иное действие относительно его текущей локальной системы координат.

Ещё одно важное свойство матричных аффинных преобразований – невырожденность. Для любой матрицы аффинного преобразования существует обратная, более того, она соответствует обратному преобразованию. Например, если T – матрица переноса, то T^{-1} – матрица, которая «вернёт на место» объект, перемещённый при помощи T. Данное свойство справедливо и для произвольной цепочки аффинных преобразований. Таким образом, любое преобразование можно «отменить», умножив (слева) вершины объекта, трансформированного матрицей M, на матрицу M^{-1}.

Поскольку положения объектов на сцене обычно динамически изменяются (например, в ответ на действия пользователя), координаты их вершин оставляют локальными, и перед каждым выводом объекта на экран применяют к ним необходимые преобразования. Частое переписывание значений в видеопамяти неэффективно, так как шина передачи данных между ОЗУ и видеопамятью является узким местом с точки зрения скорости работы. При этом умножение матрицы преобразования на координаты каждой вершины имеет аппаратную поддержку графического процессора, потому выполняется достаточно быстро. Настолько быстро, что в процессе подготовки каждого кадра за приемлемое для демонстрации плавных движений время удаётся обрабатывать сотни тысяч вершин на мобильных устройствах и миллионы вершин на настольных компьютерах.

Умножение происходит на этапе преобразования вершин графического конвейера. Каждому объекту сцены ставится в соответствие его матрица преобразования, которая называется *матрицей модели* (англ. *Model Matrix*). Она хранится вместе с дескриптором объекта в ОЗУ (предполагается, что сами данные объекта хранятся в видеопамяти, а в ОЗУ – лишь его дескриптор, необходимый для оперирования объектом на стороне приложения). Перед отрисовкой очередного кадра эта матрица изменяется так, чтобы обеспечить текущее положение объекта на сцене, и передаётся в видеопамять. Пересылка 16 значений матрицы происходит значительно быстрее,

чем копирование всего массива вершин объекта, поэтому частое её изменение не снижает производительность.

Суперпозиция аффинных преобразований путём перемножения соответствующих матриц позволяет сохранить взаимное расположение объектов, тем самым объединив их «иерархическими» связями. При этом справедливо следующее выражение:

$$M = M_p M_c,$$

где M – итоговая матрица для объекта,
M_p – матрица «родительского» объекта (относительно которого должны быть расположены «дочерние» объекты),
M_c – матрица «дочернего» (данного) объекта, обеспечивающая изменение его положения относительно его «родителя».

Например, если необходимо разместить на сцене модель стола, а на ней – модель вазы, то удобно задать матрицу преобразования для вазы, изменяющую её положение относительно начала координат стола, матрицу преобразования стола, изменяющую его положение относительно начала координат сцены, затем трансформировать стол его матрицей, а к вазе – результат перемножения, как показано выше. В этом случае появляется возможность организовать зависимость положения вазы от положения стола на сцене: ваза будет «автоматически» перемещаться вслед за столом, оставаясь относительно него неподвижной. На таком принципе можно создавать сложные иерархические конструкции и минимальными изменениями управлять их положением на сцене.

4.2. Углы Эйлера и шарнирный замок

Особого внимания заслуживает вопрос комбинирования поворотов объекта. Как было отмечено выше, ориентацию объекта в пространстве можно задать при помощи трёх поворотов вокруг взаимно перпендикулярных осей. Такой способ носит название *углов Эйлера* (англ. *Euler Angles*) и предполагает выражать ориентацию объекта в виде трёх числовых значений, соответствующих трём углам. Конкретный способ выбора осей поворота для определения углов Эй-

лера имеет множество вариаций. Наиболее часто используется способ Тэйта-Брайана (англ. *Tait-Bryan Angles*), при котором все три оси различны.

Углы поворота имеют различные названия в зависимости от конкретной предметной области: способ Тэйта-Брайана используются в теоретической механике, авиации, мореплавании, навигации. В компьютерной графике принято в данном случае использовать авиационную терминологию и называть углы так: *тангаж* (англ. *Pitch*) – поворот объекта вокруг поперечной оси; *рыскание* (англ. *Yaw*) – поворот объекта вокруг вертикальной оси; *крен* (англ. *Roll*) – поворот объекта вокруг продольной оси. Наименование углов продемонстрировано на рис. 5.

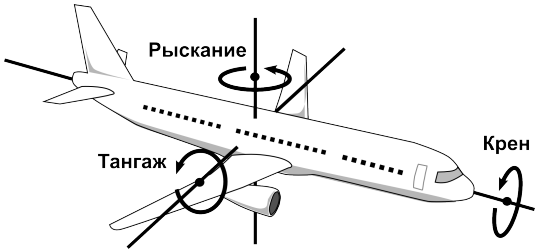

Рис. 5. Наименование углов Тэйта-Брайана

Для задания ориентации удобно положить оси тангажа, рыскания и крена совпадающими с осями локальной системы координат объекта. В этом случае можно уйти от специальных названий углов и именовать их в соответствии с осями координат: r_x, r_y, r_z.

Важным свойством поворотов, выраженных углами Эйлера (и углами Тэйта-Брайана в частности), является зависимость от порядка применения. Так, например, при фиксированных числовых значениях углов применение поворотов в последовательности r_x, r_y, r_z и в последовательности r_y, r_x, r_z дадут различные итоговые ориентации объекта. При этом не важно, выполняются ли эти повороты относительно глобальной (статичной, не поворачивающейся вместе с объектом) или же относительно локальной (меняющей ориентацию с каждым поворотом) системы координат – их порядок в обоих случа-

ях будет важен. Поэтому, при оперировании углами Эйлера, помимо задания осей, всегда оговаривают порядок применения (иерархию) поворотов.

Если оси вращения совпадают с осями локальной системы координат объекта, центр объекта совпадает с началом координат и задан некоторый порядок применения поворотов, например, r_x, r_y, r_z, то итоговая ориентация объекта может быть получена путём умножения каждой его вершины слева на матрицу вида

$$M = R_x(r_x)R_y(r_y)R_z(r_z),$$

где M – итоговая матрица для объекта,
$R_x(r_x)$ – матрица поворота вокруг вектора $(1, 0, 0)$ на угол r_x,
$R_y(r_y)$ – матрица поворота вокруг вектора $(0, 1, 0)$ на угол r_y,
$R_z(r_z)$ – матрица поворота вокруг вектора $(0, 0, 1)$ на угол r_z.

Таким образом, повороты на углы Тэйта-Брайана оказываются достаточно интуитивным способом задания ориентации объекта.

Однако этот способ, как и любой, основанный на углах Эйлера, имеет серьёзный недостаток: *шарнирный замок* (англ. *Gimbal Lock*). Шарнирный замок (складывание рамок, блокировка осей, карданная блокировка) – это ситуация потери одной из степеней свободы в процессе вращения при определённых значениях углов Эйлера.

Он возникает в момент, когда второй по иерархии поворот (в приведённом выше примере – поворот на угол r_y) составляет $\pm\pi/2$. Причина этой ситуации состоит в том, что после второго поворота локальная ось первого поворота совпадает с глобальной осью третьего поворота. В результате получается, что первый и третий повороты вращают объект вокруг одной и той же глобальной оси, то есть теряется одна из степеней свободы. Эта ситуация продемонстрирована на рис. 6.

Как можно видеть, ось X на рис. 6 (а) коллинеарна оси Z на рис. 6 (г), в результате чего повороты, изображённые на этих рисунках (первый и третий повороты в иерархии углов Эйлера) влияют на одну и ту же степень свободы объекта.

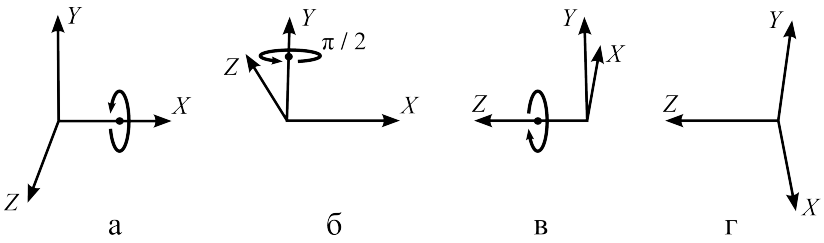

Рис. 6. Шарнирный замок: (а) – начальное положение объекта, (б) – положение после поворота на угол r_x, (в) – положение после поворота на угол $r_y = \pi/2$, (г) – положение после поворота на угол r_z

Ещё одной проблемой углов Эйлера (и углов Тэйта-Брайана в частности) является сложность анимации. Если необходимо плавно изменить ориентацию объекта по кратчайшему пути, линейная интерполяция углов Эйлера не даст корректного результата (равно как и линейная интерполяция географических координат, по сути тоже являющихся углами Эйлера, не даст кратчайший путь между двумя точками на поверхности планеты).

Решить указанные проблемы можно путём представления поворотов при помощи *кватернионов* (англ. *Quaternion*). Однако данный подход читателю предлагается изучить самостоятельно.

Часто при создании интерактивных систем визуализации требуется связать поворот объекта с командами пользователя. Пусть, например, нажатие клавиш со стрелками должно поворачивать объект в соответствующих направлениях вне зависимости от его текущей ориентации.

Если использовать углы Эйлера (изменяя соответствующий угол на некоторую величину в ответ на нажатие клавиши), требование независимости от текущей ориентации выполнено не будет: например, при повороте на π вокруг оси X, направление поворотов вокруг оси Y инвертируется.

Однако вместо аккумуляции углов и создания из них матрицы поворота, можно аккумулировать саму матрицу поворота. В начале работы программы эта матрица должна быть единичной (либо должна содержать начальный поворот объекта). Нажатие каждой клави-

ши со стрелкой должно поворачивать объект на некоторый угол Δ_ϕ вокруг глобальной оси без учёта предыдущих поворотов.

В терминах матриц это означает, что текущая матрица поворота M должна быть умножена справа на матрицу поворота $R_v(\Delta_\phi)$, составленную на основе угла Δ_ϕ и вектора v, задающего ось вращения. Вектор v необходимо выбрать таким образом, чтобы он выражал глобальную ось a желаемого поворота в системе координат, задаваемой текущей матрицей M. Это означает, что $Mv = a$, то есть $v = M^{-1}a$.

Так как M – матрица поворота, её обратная матрица совпадает с транспонированной: $M^{-1} = M^T$. Так как глобальная ось поворота a, скорее всего, совпадает с одной из координатных осей, для получения вектора v достаточно просто взять соответствующий столбец матрицы M^T (см. рис. 4), или соответствующую строку матрицы M.

Например, если нажата клавиша со стрелкой вправо и поворот должен осуществляться вокруг глобальной оси Y, новая матрица поворота определяется как

$$M' = MR_{(m_1,m_5,m_9)}(\Delta_\phi),$$

где M' – итоговая матрица поворота (которая станет текущей на следующем шаге),
M – текущая матрица поворота,
$R_{(m_1,m_5,m_9)}(\Delta_\phi)$ – матрица поворота на угол Δ_ϕ вокруг вектора, составленного из элементов 1, 5 и 9 матрицы M.

По нажатию клавиши со стрелкой влево формула вычисления матрицы остаётся прежней, но в качестве величины угла используется $-\Delta_\phi$.

Аналогично, если нажата клавиша со стрелкой вверх и поворот должен осуществляться вокруг глобальной ост X, новая матрица поворота определяется как

$$M' = MR_{(m_0,m_4,m_8)}(\Delta_\phi),$$

где $R_{(m_0,m_4,m_8)}(\Delta_\phi)$ – матрица поворота на угол Δ_ϕ вокруг вектора, составленного из элементов 0, 4 и 8 матрицы M.

4.3. Преобразования проекции

Проекция объектов на плоскость экрана производится по следующему алгоритму:

1. Выделяется фиксированная область пространства $\Xi \subset \mathbb{R}^3$ в виде параллелепипеда, называемая *нормированными координатами устройства* (англ. *Normalized Device Coordinates*, NDC).
2. На сцене выделяется «область интереса» $\Omega \subset \mathbb{R}^3$, т. е. часть пространства, которая должна быть видима в данный момент в соответствии с логикой работы графического приложения.
3. Ко всем вершинам всех объектов сцены применяется преобразование
$$p(v) = \begin{cases} v' \in \Xi, & v \in \Omega \\ v'' \in \Psi, \Psi \cap \Xi = \varnothing, & v \in \Phi, \Phi \cap \Omega = \varnothing \end{cases},$$
т. е. преобразование, которое переводит объекты из «области интереса» в NDC.
4. На экране выделяется прямоугольная область, называемая *портом просмотра* (англ. *Viewport*).
5. Осуществляется проекция NDC в порт просмотра путём отбрасывания координаты Z и линейного масштабирования итогового множества точек до совпадения с размером порта просмотра.

Конкретный вид NDC является конвенционализмом (договорённостью), принятым в каждом конкретном стандарте графического API. В OpenGL за NDC принят куб со стороной 2 и центром в начале координат, т. е. куб $[-1, 1] \times [-1, 1] \times [-1, 1]$. В Direct3D за NDC принят параллелепипед $[-1, 1] \times [-1, 1] \times [0, 1]$.

Направление осей NDC также является конвенционализмом. Отличие осей в NDC OpenGL и Direct3D представлено на рис. 7.

На основе приведённого выше алгоритма можно сформулировать общее утверждение: *на экран попадают те и только те объекты сцены, которые в результате всех преобразований попали в NDC*. Это утверждение справедливо для всех современных графических API.

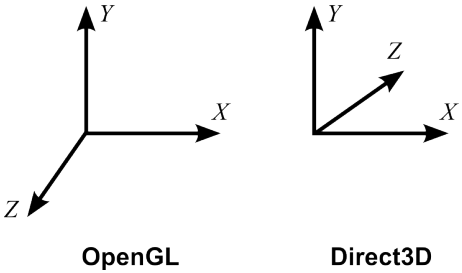

Рис. 7. Система координат, принятая в OpenGL и Direct3D

Для удобства представления преобразований отображение «области интереса» в NDC выделяют в отдельную матрицу. Эту матрицу называют *матрицей проекции* (англ. *Projection Matrix*). Однако с математической точки зрения она описывает не саму проекцию, а лишь подготовительное преобразование (умножение слева координат точки пространства на эту матрицу не понижает размерности точки, а лишь перемещает её из области интереса в NDC).

Наиболее часто используемыми в компьютерной графике проекциями являются *параллельная* (англ. *Parallel Projection*) и *перспективная* (англ. *Perspective Projection*).

Параллельная проекция, как следует из названия, сохраняет параллельность прямых линий. В этом случае «область интереса» представляет собой параллелепипед, называемый *параллелепипедом видимости* (англ. *View Piped*), который отображается на NDC при помощи обычных аффинных преобразований. Параллелепипед видимости и полученная в итоге проекция на экран показаны на рис. 8. Такой тип проекции используется в основном в чертёжном деле (так как не вносит искажений в объекты) и при отображении двумерной графики средствами трёхмерной (например, индикаторы здоровья, силы и боеприпасов персонажа, отображаемые поверх игровой сцены).

Стороны параллелепипеда называются *левой* (англ. *Left*), *правой* (англ. *Right*), *верхней* (англ. *Top*), нижней (англ. *Bottom*), *ближней* (англ. *Near*) и *дальней* (англ. *Far*) плоскостями отсечения.

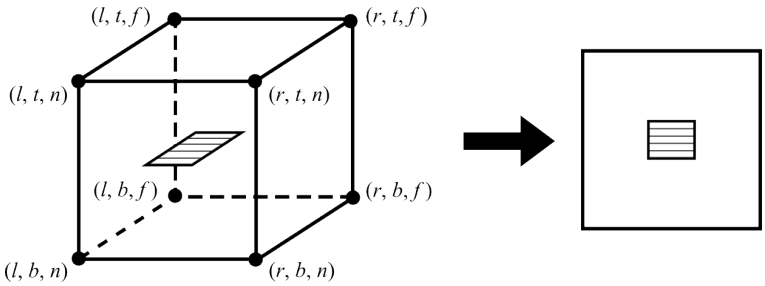

Рис. 8. Параллелепипед видимости и результат параллельной проекции

Такие названия обусловлены тем, что эти стороны, по сути, ограничивают видимую область пространства, отсекая невидимые части сцены.

Исходя из указанных на рис. 8 обозначений, матрица проекции для этого случая имеет вид

$$P_{par} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Перспективная проекция учитывает искажения, наблюдаемые в реальной жизни, когда более далёкие от наблюдателя объекты кажутся более маленькими. Чтобы смоделировать этот эффект, «область интереса» представляется в виде усечённой пирамиды, называемой *усечённой пирамидой видимости* (англ. *View Frustum*), расширяющейся с удалением от наблюдателя. При отображении такой области в параллелепипед NDC осуществляется сжатие объектов, причём коэффициент сжатия будет зависеть от удалённости. Усечённая пирамида видимости и полученная в итоге проекция на экран показаны на рис. 9. Такой тип проекции используется для реалистичной визуализации трёхмерных сцен (при создании компьютерных игр, симуляторов, компьютерной мультипликации, кинематографических эффектов и т. п.), так как соответствует восприятию человека.

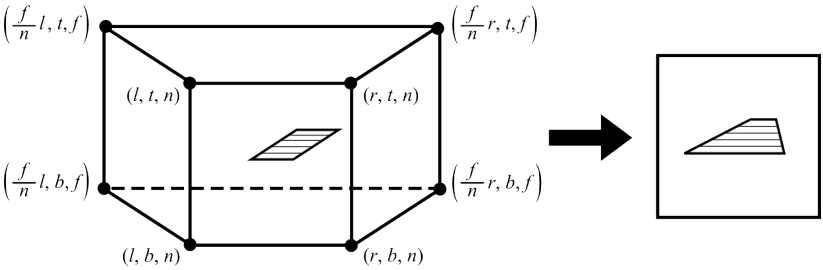

Рис. 9. Усечённая пирамида видимости и результат перспективной проекции

Преобразование «области интереса» в NDC, используемое в данном случае, не сохраняет параллельность прямых, поэтому оно по определению не может быть выражено через аффинные. Тем не менее оно допускает матричное представление вида

$$v = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}, \quad v' = P_{persp} v, \quad v'' = \frac{v'}{v'_w},$$

где v – исходный вектор,
v'' – результирующий вектор,
P_{persp} – матрица перспективной проекции.

С учётом обозначений рис. 9 матрица перспективной проекции имеет вид

$$P_{persp} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

Поскольку в случае аффинных преобразований $v_w \equiv 1$, оба типа проекции сводятся к единообразному механизму.

Для задания перспективной проекции удобно бывает пользоваться понятием *угла обзора* (англ. *Field Of View*, FOV). Смысл этого понятия отражён на рис. 10.

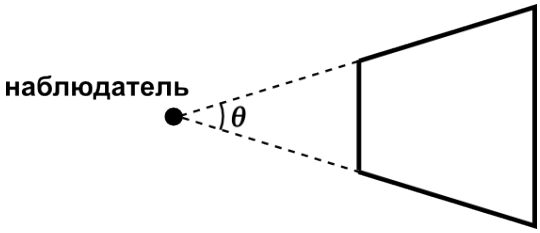

Рис. 10. Связь угла обзора θ и усечённой пирамиды видимости

В таком случае, для заданных $z = n$ и $z = f$ ближней и дальней плоскостей отсечения соответственно, элементы матрицы перспективной проекции могут быть вычислены на основе следующих соотношений:

$$-l = r = n \operatorname{tg} \frac{\theta}{2}, \quad -b = t = n \frac{w}{h} \operatorname{tg} \frac{\theta}{2},$$

где w, h – соответственно ширина и высота сторон экрана (обычно в пикселях), на который будет произведена проекция,
θ – угол обзора.

Значения n и f для перспективной проекции выбираются исходя из структуры сцены, но, как видно из приведённых формул, должно быть соблюдено условие $n \neq f \neq 0$. Для параллельной проекции это условие является более мягким: $n \neq f$.

С учётом вышесказанного, положение объекта в NDC можно получить, преобразовав все его вершины по формуле

$$v' = PMv, \quad v'' = \frac{v'}{v'_w},$$

где v'' – результирующий вектор координат вершины,
v – исходный вектор координат вершины,
M – матрица модели (для размещения объекта на сцене, возможно – результат перемножения множества матриц),
P – матрица проекции.

Как правило, нормировку вектора (деление его компонентов на w) берёт на себя графический процессор, делая это автоматически. От программиста же требуется вычислить лишь первичную трансформацию вершины, т. е. $v' = PMv$.

Для работы с матрицами в приложении могут быть использованы специальные библиотеки, например, *Eigen* [3], *OpenSceneGraph* [4] или *GLM* [5]. Эти библиотеки предоставляют необходимые структуры данных и функции для того, чтобы оперировать векторами и матрицами.

Задание: В целях тренировки читателю предлагается реализовать мини-библиотеку работы с матрицами и векторами самостоятельно. Библиотека должна позволять конструировать матрицы аффинных преобразований, матрицы проекций и векторы, осуществлять их перемножение.

Подсказка: Для эффективной работы графического приложения матрицы следует хранить в виде массива `float[16]`. Касательно векторов возможны варианты, так как часто бывают нужны векторы размерности 2 и 3, причём различных типов (как `float`, так и `int`). В связи с этим, реализуя проект на C++, имеет смысл создать шаблонный класс либо несколько классов для различных нужд.

Замечание: На сегодняшний день графические процессоры при построении сцены используют одинарную точность чисел с плавающей точкой (`float`, а не `double`). Поэтому двойная точность не нужна для представления векторов и матриц, которые будут использованы непосредственно при построении сцены (и может понадобится лишь для произведения сопутствующих расчётов, предусмотренных логикой графического приложения).

5. Структура графического приложения

Под графическим приложением будем подразумевать программу, работающую с графикой посредством прямых вызовов низкоуровневого API (такое понятие, вообще говоря, является жаргонизмом, однако активно используется в литературе).

5.1. Графический контекст

Ключевой структурой данных в графическом приложении является *графический контекст* (англ. *Graphical Context*). Он объединяет в себе дескрипторы всех графических ресурсов (например, объектов сцены, представленных в виде массивов вершин и хранящихся в видеопамяти), дескрипторы точек доступа к видеодрайверу и все необходимые для управления сценой и процессом визуализации сервисные структуры данных.

Конкретная структура графического контекста различается для разных стандартов графического API и для разных платформ. Для его создания используются механизмы, предусмотренные конкретной операционной средой (например, оконной системой). В связи с этим, работая с графическим контекстом напрямую, невозможно полностью избавиться от системно-зависимого кода, даже если в качестве стандарта вывода графики используется OpenGL, имеющий реализации почти под все существующие платформы. Однако системно-зависимый код может быть сведён к минимуму и ограничен лишь инициализацией контекста. Все дальнейшие действия по управлению и отрисовке сцены стандартизованы и от конкретной платформы уже не зависят.

Важной структурой данных, неразрывно связанной с графическим контекстом, является *поверхность рендеринга* (англ. *Rendering Surface*) – объект в составе конкретной системы отображения (оконной или иной системы управления отображением

элементов графического интерфейса приложений), при помощи которого происходит демонстрация результатов визуализации сцены.

5.2. Обобщённое графическое приложение

Большинство графических приложений интерактивны, т. е. предполагают активное участие пользователя в работе. Типичными примерами являются компьютерные и видеоигры, симуляторы, графические редакторы. В таких приложениях управление сценой полностью предоставлено пользователю, который инициирует своими командами большую часть происходящих в приложении действий. Существуют, однако, и такие графические приложения, как, например, видеоплееры. В них влияние пользователя на происходящее сводится к заданию начальных настроек (разрешение экрана, настройка звука, субтитров и т. д.), а затем лишь к приостановке/возобновлению воспроизведения и перемотке. Роль основного управляющего механизма берёт на себя сама система, используя средства слежения за временем (таймеры).

Тем не менее в обоих случаях внутренняя структура графического приложения почти идентична, а различается лишь набор допустимых для пользователя действий и реакций системы на них. Чаще всего графическое приложение содержит потенциально бесконечный цикл (прерывание которого происходит лишь по команде пользователя на завершение приложения). В этом цикле с учётом текущего состояния приложения (отданных пользователем команд) перестраивается сцена и выполняется отправка всех необходимых данных на графический конвейер. При необходимости, в конце цикла выполняется команда обновления поверхности рендеринга.

Обычно поверхность рендеринга устроена с использованием механизма *двойной буферизации* (англ. *Double Buffering*): имеется два буфера, видимый (показываемый пользователю в данный момент) и активный (подверженный изменению в процессе работы графического конвейера). Команда обновления такой поверхности рендеринга называется *сменой буферов* (англ. *Swap Buffers*). Она пред-

полагает ожидание завершения всех команд, отправленных на конвейер, и обмен ролями активного и видимого буферов.

Общую структуру графического приложения можно описать следующей последовательностью шагов:

1. Инициализировать графический контекст и поверхность рендеринга.
2. Создать графические ресурсы.
3. Построить сцену.
4. Пока не дана команда завершения, повторять:
   4.1. Изменить состояние сцены в соответствии с текущим состоянием приложения.
   4.2. Отправить данные на графический конвейер.
   4.3. Обновить поверхность рендеринга.
5. Удалить графические ресурсы.
6. Удалить графический контекст и поверхность рендеринга.

Состояние сцены включает в себя набор присутствующих на ней объектов, их положение (т. е. значения их матриц преобразования) и параметры (визуальные свойства), значения матриц проекции и параметры графического конвейера. Набор параметров графического конвейера называется *машиной состояний* (англ. *State Machine*). Эти параметры управляют работой различных этапов конвейера в пределах допустимых вариаций, описанных стандартом графического API.

Шаги (2) и (3) в указанном выше алгоритме иногда вносят в тело цикла (4). Однако создание ресурсов – затратная операция, потому рекомендуется выносить её в отдельный этап и в разумных пределах осуществлять опережающую загрузку. То же самое касается начального построения сцены (под которым понимается инициализация положения и визуальных свойств объектов).

По тем же причинам этап (4.1) следует делать как можно более коротким, не допуская «холостых» изменений состояния объектов и конвейера (многократных изменений одного и того же свойства без вызова отрисовки).

Следует обратить особое внимание на шаги (5) и (6). Хотя при завершении приложения ОС берёт на себя задачу утилизации всех

выделенных ресурсов, *хорошим тоном* является полная очистка памяти и удаление всех созданных ресурсов вручную. Это важно в случае, если написанный код затем будет перемещён в более крупный проект, куда войдёт в составе отдельного модуля.

5.3. Анимация

Цикл (4) не содержит никаких сторонних средств организации задержки и выполняется с той максимальной скоростью, какую обеспечивает система. Шаг (4.1) должен учитывать этот факт в логике своей работы, если в нём предусмотрена программная *анимация* (англ. *Animation*).

Анимация – это изменение свойств сцены (или отдельных объектов сцены) с течением времени, демонстрируемое пользователю в виде последовательности кадров, каждый из которых по возможности минимально отличается от предыдущего (сохраняется *межкадровое соответствие*, англ. *Interframe Coherence*).

Фактическая скорость перерисовки сцены может меняться в зависимости от внешних факторов, например текущей загруженности вычислительной системы. Чтобы сделать скорость анимации независимой от фактической скорости перерисовки, необходимо программно измерять время, затраченное на одну итерацию, и использовать его в качестве масштабного коэффициента для величины изменения свойств объектов. Например, если с момента предыдущего измерения прошло Δ_t секунд (в системах визуализации, работающих в реальном времени, $\Delta_t < 1$) и есть некоторый параметр сцены p, который в соответствии с логикой приложения должен линейно измениться на Δ_p за секунду, то

$$p^{(i)} = p^{(i-1)} + \Delta_p \Delta_t,$$

где i – номер итерации,
$p^{(i)}$ – значение параметра p на i-й итерации.

У монитора, при помощи которого пользователь просматривает результат визуализации, есть некоторая рабочая частота, т. е.

скорость обновления изображения. На сегодняшний день большинство мониторов работают на частоте 60 Гц, т. е. тратят на обновление $1/60$ секунды. Следовательно, приложению нет смысла генерировать больше изображений в секунду, чем способен отобразить монитор. Для удовлетворения такого ограничения используется *вертикальная синхронизация* (англ. *Vertical Synchronization*, vsync) – механизм блокировки процесса на шаге (4.3) до готовности монитора.

На этапе отладки вертикальную синхронизацию можно отключить, чтобы программно оценить скорость выполнения подготовки изображения. Однако в итоговом продукте вертикальная синхронизация должна быть включена, чтобы приложение не совершало «холостых» итераций.

В случае отображения сложных сцен, скорость подготовки изображения может оказаться меньше скорости обновления монитора. В идеале это означает необходимость оптимизации кода и графических ресурсов, однако на практике оптимизация не всегда приводит к желаемому результату. Именно поэтому для каждого графического приложения формулируют минимальные требования к конфигурации вычислительной системы, при удовлетворении которых приложение гарантированно может обеспечить *приемлемую* для пользователя скорость визуализации. На сегодняшний день стандартом де-факто «идеальной» скорости является 60 кадров в секунду (англ. *Frames Per Second*, FPS). Однако *приемлемыми* значениями можно считать и 30, и даже 20 FPS. Согласно исследованиям минимальная скорость для относительно комфортного восприятия движений человеком – 8 FPS [6].

5.4. Многопоточность графического приложения

Отдельного внимания заслуживает вопрос многопоточности графического приложения. С целью оптимизации нагрузки вычислительной системы имеет смысл вынести цикл (4) в отдельный поток, в особенности, если обрабатываемая сцена достаточно сложна. Идея в том, чтобы отделить визуальную часть от обработки команд пользо-

вателя (которая обычно выполняется в главном потоке приложения), тем самым обеспечив большую отзывчивость приложения.

В зависимости от конкретной платформы и конкретного графического API контекст может быть связан либо с процессом, либо с потоком. Например, в ОС Windows, GNU/Linux и OS X контекст OpenGL связан с процессом, и обращение к нему (т. е. вызов API OpenGL) может быть осуществлён из любого потока. В iOS, напротив, контекст связан с потоком, а для разделения одних и тех же графических ресурсов между разными потоками существует специальный механизм объединения нескольких контекстов в группу.

Одни стандарты графических API предусматривают встроенные средства потокобезопасного использования, но другие – нет. Так, например, в Direct3D есть возможность осуществлять одновременные обращения к API функциям из разных потоков, а OpenGL требует организации монопольного доступа к графическим ресурсам. При этом, например, в системе OS X в контекст OpenGL встроен мьютекс, обеспечивающий синхронизацию, но захват и освобождение этого мьютекса программист должен осуществлять в явном виде. В системе iOS, напротив, в контексте OpenGL (точнее, OpenGLES, так как в iOS используется версия OpenGL для мобильных устройств) встроенных средств синхронизации нет, и для организации монопольного доступа к общим графическим ресурсам программисту требуется оформлять критические секции полностью самостоятельно.

В целях упрощения подачи материала в данном пособии на практике не рассматриваются вопросы организации многопоточных графических приложений. Во всех приведённых примерах вся работа с графическими ресурсами, включая их загрузку и изменение, будет производится в главном потоке приложения.

5.5. Буфер кадра

В процессе работы графического конвейера порождается некоторое множество данных, которые необходимо сохранять для дальнейшего использования. Это не только финальное изображение, но и различная сервисная информация, которая может быть нужна

как самому конвейеру, так и приложению (для реализации дополнительной логики). Порождаемые данные сохраняются в специальные структуры, называемые буферами. Как правило, каждый буфер представляет собой матрицу точек, по размеру равную итоговому растру, структура элементов которой зависит от назначения буфера. Например, для сохранения итогового изображения используется *буфер цвета* (англ. *Color Buffer*), в ячейках которого сохраняются цвета, закодированные в соответствии с некоторой цветовой моделью (чаще всего RGB или RGBA). Другие буферы (служащие для хранения сервисной информации) будут рассмотрены позднее.

За частью буферов чётко закреплено их назначение. Другие же могут быть использованы для сохранения произвольной информации, вычисляемой на конвейере, которая может быть нужна приложению для организации различных специальных эффектов. Такие буферы называются *целями рендеринга* (англ. *Render Target*). Программист сам определяет число и назначение целей рендеринга, а также принцип их заполнения.

Перед началом работы, как правило, выполняется очистка каждого из буферов, т. е. заполнение их некоторым начальным набором значений.

Все используемые буферы объединяются в одну агрегирующую структуру данных, называемую *буфером кадра* (англ. *Frame Buffer*). В одном приложении может быть произвольное число буферов кадра, которые, как правило, создаются на этапе подготовки графических ресурсов, а затем переключаются по необходимости (в соответствии с логикой приложения при подготовке каждого следующего кадра). В каждый момент времени активным буфером кадра может быть только один.

5.6. Первое приложение

Изучение низкоуровневых средств работы с графикой предлагается начать со стандарта OpenGL. Данный стандарт описывает лишь работу с графикой (в рамках приведённого ранее графического конвейера), и ничего более. Операции, связанные с созданием окна,

обработкой команд пользователя, загрузкой внешних мультимедийных ресурсов (изображений и 3D-моделей) и т. п. этим стандартом не специфицируются. Для упрощения создания графического контекста, поверхности рендеринга, обработки команд пользователя и отображения результата под управлением конкретной ОС предлагается использовать мультиплатформенные библиотеки GLEW [7] и GLFW [8].

Библиотека GLEW предназначена для упрощения этапа динамической загрузки функций библиотеки, реализующей стандарт OpenGL. Библиотека GLFW предназначена для упрощения этапа создания окна с контекстом OpenGL и поверхностью рендеринга (независимым от конкретной ОС способом).

Следует отметить, что стандарт OpenGL прошёл достаточно длинный путь развития, в ходе которого в нём, вместе с развитием элементной базы графических процессоров, появлялись и исчезали различные функции. В данном пособии рассматривается OpenGL версии 3.3+ (для настольных компьютеров). На сегодняшний день эта версия полностью поддерживается большинством настольных компьютеров под управлением всех популярных ОС. На мобильных устройствах под управлением iOS и Android используется OpenGLES 2.0 и 3.0, «облегчённая» модификация OpenGL (ввиду ограничений аппаратуры). Для создания графических Web-приложений используется WebGL, основанный на OpenGLES 2.0 (для организации совместимости и с настольными компьютерами, и с мобильными устройствами). Однако в отношении общей логики работы настольная, мобильная и Web-ориентированная версии эквивалентны. Таким образом, если читатель освоит наиболее полнофункциональный стандарт (настольную версию), ему не составит труда в дальнейшем перейти и на облегчённые.

Заготовку приложения (на языке C++), на основе которой можно будет реализовывать описанные далее методы и алгоритмы, можно скачать в сети Интернет по адресу https://github.com/icosaeder/cg-tutorial.git.

В целях увеличения наглядности код заготовки разбит на функции, соответствующие шагам приведённого выше общего алгоритма работы графического приложения.

Стандарт OpenGL предполагает чисто процедурный стиль программирования (хотя имеет биндинги в самые разные языки программирования, включая объектно-ориентированные и функциональные). Поэтому все графические ресурсы (например, объекты сцены) в нём представлены в виде целочисленных знаковых или беззнаковых дескрипторов. Для модификации каждого из них используется «политика активации»: существуют специальные функции, принимающие в качестве параметра дескриптор и делающие соответствующий ресурс активным, а также функции модификации каких-либо параметров активного ресурса.

Следует обратить внимание на то, что стандарт OpenGL и вспомогательные библиотеки (такие как GLEW и GLFW) придерживаются единого соглашения о именовании функций, типов данных и констант. Все функции и типы названы в *«верблюжьем регистре»* (англ. *Camel Case*), начиная с префикса, отражающего их принадлежность к конкретной библиотеке. У функций префикс пишется маленькими буквами, у типов – большими. Константы записаны в *«заглавном змеином регистре»* (англ. *Capitalized Snake Case*). Например, функция очистки буферов по маске называется `glClear`, тип маски: `GLbitfield`, константа маски буфера цвета: `GL_COLOR_BUFFER_BIT`.

Работа графического конвейера OpenGL управляется машиной состояний. Она представляет собой набор свойств, значения которых можно изменять при помощи специальных функций. Бинарные свойства изменяются функциями `glEnable` (для включения) и `glDisable` (для выключения), принимающими в качестве параметра константу, именующую свойство. Небинарные свойства изменяются специальными функциями, как, например, `glClearColor`, которая устанавливает значение для очистки буфера цвета.

Машина состояний, так же как и графические ресурсы, связана с контекстом. Таким образом, функции изменения машины состояний, так же как и функции активации и модификации ресурсов,

могут быть использованы в любом месте в приложении при условии наличия активного контекста.

Приложение по указанной выше ссылке лишь создаёт окно с контекстом и поверхностью рендеринга OpenGL и закрашивает его синим цветов. Закраска достигается путём очистки буфера цвета. Буфер цвета создаётся автоматически при создании контекста как часть *буфера кадра по умолчанию* (англ. *Default Frame Buffer*). В случае данного приложения (где создание контекста берёт на себя GLEW) ссылки на буфер кадра по умолчанию в явном виде нет. Тем не менее он выставлен в качестве активного, поэтому все операции по изменению – установка цвета для очистки, команда очистки и т. д. – касаются именно его.

Задание: Подготовить рабочее пространство для дальнейшего изучения стандарта OpenGL: настроить компилятор C++ (или среду программирования) так, чтобы он успешно собрал описанное выше приложение.

Подсказка: Требуется установить библиотеки GLEW, GLFW и OpenGL на используемую читателем операционную систему и настроить доступ компилятора/линковщика к ним. В UNIX-подобных операционных системах установка может быть совершена при помощи используемого в конкретной ОС пакетного менеджера.

6. Шейдеры

На первых этапах развития компьютерной графики конвейер был *фиксированным* (англ. *Fixed Pipeline*), т. е. каждый из его этапов был реализован в соответствии с определённым наперёд заданным алгоритмом, а настройка конвейера производилась лишь изменением ограниченного набора параметров этих алгоритмов. Но современное графическое оборудование поддерживает полное переопределение некоторых этапов конвейера. Таким образом, фиксированный конвейер уступил место *программируемому* (англ. *Programmable Pipeline*).

Программируемый конвейер является значительно более гибким инструментом для генерации изображений, поэтому современные графические системы в большинстве своём используют именно его. Каждый из этапов конвейера переопределяется в отдельности путём подстановки на его место исполняемого модуля, называемого *шейдером* (англ. *Shader*).

Программируемыми являются не все этапы: в зависимости от графического оборудования и используемого графического API часть этапов должна быть определена программистом обязательно (не имеет реализации по умолчанию), часть имеет реализацию по умолчанию, но допускает переопределение, а часть фиксирована и управляется лишь машиной состояний.

6.1. Свойства шейдеров

Шейдеры иногда называют *микропрограммами* для этапов графического конвейера. Здесь приставка «микро» подчёркивает их малый размер: шейдер выполняет лишь узкоспециализированную задачу и для увеличения производительности конвейера должен быть по возможности освобождён от лишних действий. Конкретная задача и свойства шейдера определяются этапом конвейера, для которого он является исполнителем. Однако для всех шейдеров можно определить ряд общих свойств:

1. Автономность. Шейдер не является частью кода основного приложения и минимально зависит от него. Часто шейдеры входят в состав графического приложения в виде исходных кодов и компилируются во время выполнения непосредственно перед встраиванием в конвейер.
2. Аппаратная поддержка. На современном оборудовании шейдеры выполняются на видеокарте, а их исполняемый модуль автоматически оптимизируется под архитектуру GPU.
3. Специальный язык программирования. Шейдеры пишутся на некотором языке программирования, чаще всего в процедурной парадигме.
4. Параллелизм по данным. Как правило, шейдеры в явном виде не поддерживают многопоточности, однако должны быть реентерабельны на уровне всего своего кода, так как рассчитаны на многократный параллельный запуск с разными входными данными.
5. Узкая специализация. Шейдер выполняет лишь один небольшой шаг на пути построения итогового изображения, его код должен быть минимален и очень хорошо оптимизирован, так как для получения одного кадра каждый шейдер, как правило, запускается многократно (на некоторых сценах число вызовов одного шейдера на кадр может достигать миллионов).

Превращая фиксированный конвейер в программируемый, шейдеры обеспечивают целый ряд преимуществ:

1. Единообразный механизм представления разнородных визуальных эффектов.
2. Высокую эффективность графического конвейера.
3. Настраиваемость графического конвейера (вплоть до организации адаптивности, когда в зависимости от конкретной конфигурации системы для одних и тех же эффектов автоматически выбираются шейдеры различной сложности, обеспечивая тем самым учёт возможностей конкретного оборудования).
4. Децентрализацию кода.

5. Возможность переиспользования визуальных эффектов в различных приложениях (так как шейдеры, как уже отмечалось, минимально зависят от использующего их приложения).

Жизненный цикл шейдера может несколько отличаться в разных графических API, однако различия незначительны, так как этот цикл обусловлен аппаратной организацией GPU:

1. Загрузка исходного кода из файла или строковой константы.
2. Компиляция «на лету» (т. е. во время выполнения основного приложения).
3. Встраивание в конвейер (*активация*).
4. Множественное выполнение (шейдер выполняется ровно один раз для каждой сущности, обрабатываемой данной ступенью конвейера во время рендеринга кадра).
5. Отсоединение от конвейера (*деактивация*).
6. Удаление из памяти.

6.2. Виды шейдеров

Как уже отмечалось выше, шейдеры классифицируются по соответствующей им ступени графического конвейера, и число поддерживаемых типов шейдеров зависит от конкретного графического оборудования.

В данном пособии будут рассмотрены лишь два вида шейдеров, которые поддерживаются на сегодняшний день почти на любом оборудовании (включая мобильные устройства) – *вершинные шейдеры* (англ. *Vertex Shader*) и *фрагментные шейдеры* (англ. *Fragment Shader*).

Схематично место указанных шейдеров в графическом конвейере показано на рис. 11.

Вершинный шейдер – это микропрограмма, которая за один свой вызов обрабатывает одну вершину (из всего множества отправленных в данный момент на графический конвейер). Задача вершинного шейдера – определить итоговые характеристики вершины, имеющие значение для последующих этапов конвейера (например, положение вершины в пространстве).

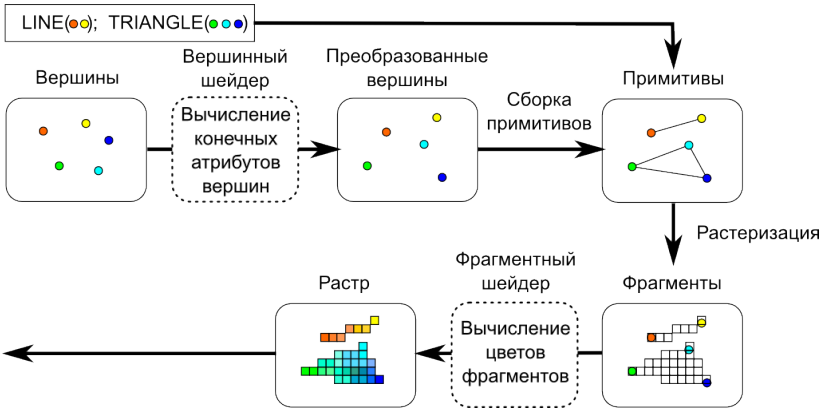

Рис. 11. Место вершинного и фрагментного шейдеров в графическом конвейере

Фрагментный шейдер – это микропрограмма, которая за один свой вызов обрабатывает один фрагмент изображения, вычисляя его числовые характеристики (например, цвет). Фрагмент – это *атомарная* ячейка растра, принадлежащая объекту сцены. Значение числовых характеристик соответствующей ячейки может быть перевычислено несколько раз в течение подготовки изображения из-за того, что объекты сцены могут накладываться друг на друга. Графический процессор автоматически подбирает фрагменты, соответствующие обрабатываемому в данный момент примитиву, и вызывает для каждого из них фрагментный шейдер.

Следует отметить, что понятие фрагмента близко к понятию пикселя, но имеет два важных отличия:

1. Число фрагментов в общем случае не равно числу пикселей, так как пиксель – ячейка итогового изображения, а фрагмент – ячейка растра, принадлежащая примитиву. В случае перекрывающихся примитивов одна и та же ячейка растра будет соответствовать разным фрагментам. В случае наличия мест на экране, не занятых объектами, не всем ячейкам будут соответствовать фрагменты.
2. Фрагмент – всегда низкоуровневое физическое понятие, атомарная ячейка растра, тогда как пиксель может быть понятием

логическим. Например, в контексте ретина-дисплеев [9] один пиксель состоит из четырёх и более фрагментов для организации *субпиксельной точности* изображений.

6.3. Язык GLSL

Для написания шейдеров используются специализированные языки программирования, которые, как правило, характеризуются следующими свойствами:

1. Процедурной парадигмой.
2. Полнотой по Тьюрингу.
3. Наличием специализированных типов данных и встроенных функций для работы с обрабатываемыми шейдером сущностями.
4. С-подобным синтаксисом.

Каждый стандарт графического API описывает свой язык для разработки шейдеров (иногда даже несколько допустимых языков). В данном пособии будет рассмотрен язык GLSL (англ. *Graphics Library Shading Language* – язык шейдеров графической библиотеки), являющийся частью стандарта OpenGL.

Как и стандарт OpenGL, язык GLSL имеет на сегодняшний день несколько версий, появившихся в результате его развития, и от версии к версии в языке происходили ощутимые изменения. Более того, стандарт OpenGLES описывает свою модификацию этого языка – ESSL (англ. *Embedded Systems Shading Language* – язык шейдеров для встраиваемых систем). В данном пособии будут рассматриваться примеры GLSL версии 330 и выше, но будут также даны комментарии по совместимости с ESSL.

Основной особенностью языка GLSL является наличие специальных типов данных и встроенных функций, облегчающих решение частых графических задач.

Базовыми типами данных являются скаляры `bool`, `int` и `float`. Числа с плавающей точкой двойной точности в общем случае не поддерживаются.

В языке ESSL для базовых типов `float` и `int` присутствуют *квалификаторы точности* (англ. *Precision Qualifiers*): `highp` (высокая точность), `mediump` (средняя точность) и `lowp` (низкая точность). Конкретные параметры точности зависят от GPU. Выбранная точность определяет диапазон представимых значений. В языке GLSL квалификаторы точности отсутствуют, и точность по умолчанию полагается максимальной.

Из базовых типов строятся двух-, трёх- и четырёхкомпонентные векторы: `bvec2`, `bvec3`, `bvec4` – векторы с компонентами типа `bool`; `ivec2`, `ivec3`, `ivec4` – векторы с компонентами типа `int`; `vec2`, `vec3`, `vec4` – векторы с компонентами типа `float`.

Для векторов перегружены операции сложения, вычитания, умножения и деления. Все эти операции выполняются *покомпонентно*, за один такт графического процессора. Следует обратить внимание, что запись вида `c = a * b;`, где `a`, `b`, `c` – векторы одинаковой размерности, приведёт к покомпонентному перемножению (хотя в геометрии и линейной алгебре такого рода умножение для векторов не определено). Это же справедливо и относительно деления.

Инициализация векторов осуществляется при помощи перегруженных конструкторов, например:

```
vec3 a = vec3(0.1, 0.2, 0.3);
vec4 b = vec4(a, 0.4); // b = {0.1,0.2,0.3,0.4}
vec2 c = vec2(a); // c = {0.1,0.2}
vec3 d = vec3(1.0); // d = {1.0,1.0,1.0}
```

Доступ к элементам можно осуществить как при помощи индекса (как у обычных массивов), так и при помощи *мнемонических полей*. Имеется три *эквивалентных* набора мнемонических полей: `xyzw`, `rgba` и `stpq`. Они обеспечивают доступ к соответствующим компонентам вектора. Технического различия в том, какие поля использовать, нет; разное написание используется исключительно с целью увеличения читаемости кода (чтобы подчеркнуть логическое назначение вектора). Записи следующего вида эквивалентны:

```
vector[1] = 1.0;
vector.y = 1.0;
```

```
vector.g = 1.0;
vector.t = 1.0;
```

Для получения субвектора либо изменения порядка компонентов могут (и с целью обеспечения эффективной работы должны) быть использованы т. н. *коктейли* (англ. *Swizzling*) – одновременная запись множества мнемонических полей, идущих в определённом порядке. Например, для извлечения из данного вектора двухкомпонентного, содержащего его компоненты y и z, должна быть использована запись вида

```
vec2 vector2 = vector1.yz;
```

Коктейли обеспечивают не только чтение, но и присваивание значений. Например, чтобы изменить порядок следования значений первых трёх компонентов вектора на обратный, можно воспользоваться записью

```
vector.xyz = vector.zyx;
```

Использование коктейлей рекомендуется всюду, где это возможно, так как оно позволяет оптимизировать доступ к векторным данным на этапе компиляции шейдера. Учитывая то, что графический процессор предназначен для эффективной работы с векторами, их использование также рекомендуется во всех возможных случаях.

Следующими важными типами языка GLSL являются четырёх-, девяти- и шестнадцатикомпонентные матрицы `mat2`, `mat3`, `mat4` с элементами типа `float`. Для них, так же как и для векторов, перегружены операции сложения, вычитания (работают покомпонентно) и умножения (работает по алгебраическим правилам). Операция деления не поддерживается. Кроме того, реализована перегрузка умножения матрицы на вектор, работающая по алгебраическим правилам и интерпретирующая вектор как столбец.

С точки зрения хранения матрица – это массив векторов-столбцов, т. е. к её элементам доступ может быть осуществлён по правилам адресации в двумерном массиве.

В GLSL поддерживается описание массивов и структур в полном соответствии со стандартом ANSI C. Элементами массивов и полями структур могут выступать любые поддерживаемые языком типы.

Для описания функций, не возвращающих никакого значения, используется тип `void`. Допустимо использование глобальных переменных (доступ к которым есть из всех функций). Однако нет ни ссылочного типа, ни указателей (работа с динамической памятью не поддерживается).

Управляющие структуры языка GLSL (циклы, условия, вызовы функций и т. п.) идентичны описанным в стандарте ANSI C.

Для удобного и эффективного решения наиболее часто встречающихся математических задач в GLSL имеется целый ряд встроенных функций. Наиболее важные и часто используемые функции приведены ниже.

1. Функции работы с векторами (есть версии функций для векторов всех размерностей):
   - 1.1. `dot(v1, v2)` – скалярное произведение `v1` и `v2`.
   - 1.2. `cross(v1, v2)` – векторное произведение `v1` и `v2`.
   - 1.3. `normalize(v)` – нормировка `v`.
   - 1.4. `reflect(v1, v2)` – отражение `v1` относительно `v2`.
   - 1.5. `refract(v1, v2, r)` – преломление `v1` относительно `v2` с коэффициентом преломления `r`.
   - 1.6. `length(v)` – длина `v`.
   - 1.7. `distance(v1, v2)` – расстояние между точками `v1` и `v2`.
2. Функции работы с матрицами (есть версии функций для матриц всех размерностей):
   - 2.1. `determinant(m)` – определитель `m`.
   - 2.2. `transpose(m)` – транспонирование `m`.
   - 2.3. `inverse(m)` – обратная `m`.
3. Тригонометрические функции (есть векторные версии функций, работающие покомпонентно):
   - 3.1. `sin(a)`, `cos(a)`, `tan(a)` – прямые функции от угла `a`, заданного в радианах.
   - 3.2. `asin(x)`, `acos(x)`, `atan(x)` – обратные функции от аргумента `x`, возвращающие угол в радианах.
   - 3.3. `radians(a)`, `degrees(a)` – перевод угла `a` из градусов в радианы и обратно.

4. Гиперболические функции (есть векторные версии функций, работающие покомпонентно):
   4.1. `sinh(x)`, `cosh(x)`, `tanh(x)` – прямые функции от аргумента `x`.
   4.2. `asinh(x)`, `acosh(x)`, `atanh(x)` – обратные функции от аргумента `x`.
5. Прочие математические функции (есть векторные версии функций, работающие покомпонентно):
   5.1. `pow(x, y)` – возведение `x` в степень `y`.
   5.2. `exp(x)` – экспонента `x`.
   5.3. `log(x)` – натуральный логарифм `x`.
   5.4. `sqrt(x)` – квадратный корень `x`.
   5.5. `abs(x)` – модуль `x`.
   5.6. `sign(x)` – сигнум `x`.
   5.7. `ceil(x)`, `floor(x)`, `round(x)` – округление `x` в разные стороны.
6. Функции выбора (есть векторные версии функций, работающие покомпонентно):
   6.1. `min(x, y)`, `max(x, y)` – минимум и максимум из `x` и `y`.
   6.2. `clamp(x, minVal, maxVal)` – ограничение `x` снизу по `minVal` и сверху по `maxVal`.
   6.3. `mix(x, y, t)` – линейная интерполяция от `x` до `y` по `t`.
   6.4. `smoothstep(x, y, t)` – гладкая эрмитова интерполяция от `x` до `y` по `t`.
   6.5. `step(e, x)` – пороговая функция, работающая по принципу

$$\text{step}(e, x) = \begin{cases} 0, & e > x, \\ 1, & e \leq x \end{cases}.$$

Полный список поддерживаемых операций, так же как и более подробную справку по GLSL, можно получить в официальной документации [10].

6.4. Взаимодействие шейдеров

При запуске шейдеры получают доступ на чтение к данным той сущности, с которой они работают. Помимо этого, однако, между шейдером и основным приложением или между шейдерами смежных этапов конвейера может быть протянута односторонняя информационная связь: основное приложение может сообщать дополнительную информацию шейдерам, а шейдеры более ранних этапов конвейера могут передавать данные шейдерам более поздних этапов.

Данные вершин называются *атрибутами* (англ. *Attribute*). Они доступны в вершинном шейдере. Для их описания в языке GLSL используется ключевое слово `in`, а в языке ESSL – ключевое слово `attribute`.

Например, в GLSL описание выглядит так:

```
in vec3 a_position;
```

В ESSL этому соответствует:

```
attribute vec3 a_position;
```

Семантически такая запись описывает один трёхкомпонентный атрибут вершины, названный `a_position`.

Следует отметить, что шейдер не обязан описывать все имеющиеся у вершины атрибуты: он может работать только с каким-то их подмножеством. Однако нужно избегать ситуаций, когда в шейдере присутствует описание атрибутов, фактически отсутствующих в вершине, так как это может привести к некорректным результатам из-за влияния неинициализированных переменных.

Число атрибутов вершин, поддерживаемых графическим процессором, ограничено, но достаточно велико, чтобы покрыть большую часть потребностей программистов.

Атрибуты предлагается именовать, начиная с префикса `a_`, чтобы подчеркнуть их назначение в программе.

Для того чтобы связать конкретные области массива данных вершин с атрибутами, описанными в вершинном шейдере, в основном приложении необходимо получить их дескрипторы. В случае использования ESSL единственной возможностью получить дескриптор является использование функции `glGetAttribLocation`, при-

нимающей в качестве аргумента символьное имя атрибута (например, `"a_position"`) и возвращающей его дескриптор. В GLSL нормальной практикой является задание дескрипторов в самом шейдере в явном виде при помощи ключевого слова `layout`.

Например, нижеприведённая запись означает, что указанному атрибуту назначен дескриптор 0:

```
layout(location = 0) in vec4 a_position;
```

Следует обратить внимание, что у каждого атрибута дескриптор должен быть уникальным.

Вершинный шейдер в результате своей работы должен как минимум означить системную переменную `gl_Position` типа `vec4`, записав туда однородные координаты соответствующей вершины после перемножения со всей имеющейся у соответствующего объекта сцены цепочкой матриц. Значение этой переменной используется на дальнейших этапах конвейера.

Матрицы преобразования, как правило, попадают в шейдер из основной программы (как часть «настроечной» информации, в которую могут входить самые разные параметры, например скорость анимации, коэффициенты плавности переходов и т. п.). Для этого используются специальные переменные, называемые *юниформами* (англ. *Uniform*). Они описываются при помощи ключевого слова `uniform`, например

```
uniform mat4 u_mvp;
```

Юниформы могут быть описаны как в вершинном, так и во фрагментном шейдере, таким образом оба типа шейдеров могут получать из объемлющей программы настроечную информацию. Их предлагается именовать, начиная с префикса `u_`, чтобы подчеркнуть их назначение в программе.

Значение каждого юниформа должно быть задано в основной программе. Для этого, так же как и в случае атрибутов, используются дескрипторы. Аналогично, при использовании ESSL или GLSL версии ниже 430 единственным способом получения дескриптора юниформа является функция `glGetUniformLocation`, принимающая символьное имя юниформа и возвращающая его дескриптор.

В GLSL версии 430 и выше есть возможность задать дескриптор вручную:

```
layout(location = 0) uniform mat4 u_mvp;
```

При этом, однако, как и в случае атрибутов, необходимо помнить об уникальности дескрипторов. Уникальность должна быть соблюдена в пределах всего конвейера, т. е. если вершинный и фрагментный шейдер имеют семантически одинаковые юниформы (например, отвечающий за один и тот же параметр алгоритма, используемого в обоих шейдерах), дескрипторы этих юниформов тем не менее должны различаться.

Для взаимодействия друг с другом шейдеры используют ещё один тип переменных, называемых *вэриинги* (англ. *Varying*). Для их описания в языке ESSL используется ключевое слово `varying`, а в языке GLSL – связка `out` в «отправляющем» шейдере и `in` в «принимающем». Важно отметить, что число и тип выходов вершинного и входов фрагментного шейдера должны совпадать, иначе конвейер не сможет функционировать.

Пример описания вэриинга в вершинном и фрагментном шейдерах на ESSL:

```
varying vec3 v_color;
. . .
varying vec3 v_color;
```

Аналогичный пример для GLSL:

```
out vec3 v_color;
. . .
in vec3 v_color;
```

Чаще всего фрагментный шейдер запускается большее число раз, чем вершинный, поскольку одному треугольнику, имеющему всего три вершины, может соответствовать целое множество фрагментов. При этом с каждым фрагментом связывается некоторое значение каждого вэриинга, отправленного вершинным шейдером в результате обработки соответствующих вершин. Значение, приходящее в вэриинг фрагментного шейдера есть результат билинейной интерполяции значений соответствующего вэриинга вершинного шейдера, полученных в результате обработки вершин растеризуе-

мого в данный момент примитива. Интерполяция осуществляется с учётом перспективных искажений примитива (которые могут возникнуть в случае перспективной проекции), что гарантирует правильное изменение параметра вдоль поверхности примитива. Функция интерполяции имеет линейный характер и не может быть переопределена.

Вэриинги предлагается именовать, начиная с префикса `v_`, чтобы подчеркнуть их назначение в программе.

Конечной целью фрагментного шейдера является определение данных для всех целей рендеринга. В ESSL цель рендеринга по умолчанию (буфер цвета) отображена на системную переменную `gl_FragColor` типа `vec4` (выражающую цвет фрагмента в RGBA). В GLSL системной переменной цвета фрагмента нет, и программист должен задать отображение на цель рендеринга в явном виде вручную путём объявления выходной переменной при помощи ключевого слова `out`:

```
layout(location = 0) out vec4 o_color;
```

Нулевой дескриптор здесь означает цель рендеринга по умолчанию (буфер цвета). Если используется лишь цель рендеринга по умолчанию, в принципе, `layout(location = 0)` можно опустить.

Выходные переменные, отображаемые на цели рендеринга, предлагается именовать, начиная с префикса `o_`, чтобы подчеркнуть их назначение в программе.

6.5. Шейдерная программа

После того как шейдер написан, он должен быть скомпилирован. По спецификации OpenGL компиляция шейдеров происходит «налету», т. е. во время выполнения основного приложения. Поскольку код шейдера невелик, компиляция происходит относительно быстро (счёт идёт на миллисекунды), тем не менее этот этап рекомендуется выносить в блок подготовительной работы (создания и загрузки графических ресурсов), т. е. отделить от процесса визуализации сцены. В процессе компиляции могут возникнуть ошибки (как при компиляции любой программы), как правило связанные с синтакси-

сом. Наличие ошибок не позволяет далее работать с данным шейдером, поэтому успешность компиляции обязательно должна проверяться, а содержание возникших ошибок имеет смысл выводить в какой-либо выходной поток (например, `stderr`).

Согласно спецификации OpenGL для встраивания в конвейер шейдеры всех этапов предварительно собираются в единый объект, называемый *шейдерной программой* (англ. *Shader Program*). На этапе сборки (*линковки*, англ. *Linking*) шейдерной программы также могут возникнуть ошибки, связанные с неверным набором шейдеров или несовпадение выходов шейдеров более ранних этапов конвейера со входами шейдеров более поздних этапов. Успешность линковки также необходимо проверять и при возникновении ошибок логировать их.

Собранная шейдерная программа должна быть активирована до отправки данных на конвейер.

Обычно в графическом приложении используется много различных шейдерных программ для обеспечения всех необходимых эффектов, и перед отправкой очередной порции данных активную программу переключают. Однако в один момент времени активной может быть только одна программа, шейдеры из которой должны полностью определять алгоритм конвейера.

6.6. Реализация

Предлагается расширить тестовое приложение, добавив в него вершинный и фрагментный шейдеры. Для упрощения запишем код шейдеров в строковые константы, а не в отдельные файлы. Следует отметить, что такой способ хранения шейдеров не всегда плох даже в реальных проектах: часто отсутствие дополнительных внешних ресурсов является желательным.

Для создания шейдера и шейдерной программы удобно завести отдельные функции. Начнём с создания шейдера. Предлагается следующая сигнатура:

```
GLuint createShader(const GLchar *code, GLenum type)
```

– `code` – строка, содержащая код шейдера;

– `type` – константа, определяющая тип создаваемого шейдера (в данном случае – вершинный или фрагментный);
– возвращаемое значение – ненулевой дескриптор шейдера, если создание прошло успешно; 0, если при создании произошла ошибка.

Тело функции приведено в листинге 1.

Листинг 1. Функция создания шейдера

```
GLuint createShader(const GLchar *code, GLenum type)
{
    GLuint result = glCreateShader(type);

    glShaderSource(result, 1, &code, NULL);
    glCompileShader(result);

    GLint compiled;
    glGetShaderiv(result, GL_COMPILE_STATUS,
                  &compiled);
    if (!compiled)
    {
        GLint infoLen = 0;
        glGetShaderiv(result, GL_INFO_LOG_LENGTH,
                      &infoLen);
        if (infoLen > 0)
        {
            char infoLog[infoLen];
            glGetShaderInfoLog(result, infoLen,
                               NULL, infoLog);
            cout << "Shader compilation error" <<
                    endl << infoLog << endl;
        }
        glDeleteShader(result);
        return 0;
    }
    return result;
}
```

В строке (3) происходит создание шейдерного объекта заданного типа (объект в данном случае – это структура данных, связанная с графическим контекстом OpenGL, а *не* экземпляр какого-либо класса, поскольку OpenGL является *полностью императивным*). Система резервирует память под управляющие структуры данных и возвращает дескриптор в виде целого беззнакового числа. При этом валидными являются ненулевые дескрипторы, ноль же означает ошибку создания. В строке (5) загружается массив строк исходного кода. Приведённая функция работает со всем исходным кодом как с одной строкой. В строке (6) отдаётся команда на компиляцию шейдера. В строке (9) запрашивается статус компилятора, чтобы проверить, не возникли ли ошибки. Если ошибки имеются, в строках (13–25) запрашивается и выводится на консоль их описание, затем созданный шейдер удаляется и возвращается 0. В противном случае возвращается дескриптор созданного и успешно скомпилированного шейдера.

Следует отметить, что как и во многих других стандартах и спецификациях, в OpenGL принято соглашение именовать функции, создающие что-либо, используя слово `Create`, а функции, запрашивающие что-либо, созданное другими, – используя слово `Get`. При этом ответственность за объект *всегда* остаётся на создателе, т. е. тот, кто создал какой-либо объект, обязан затем уничтожить его при помощи соответствующей функции, содержащей в названии слово `Delete`, а тот, кто запросил какой-то объект, должен просто забыть о нём по ненадобности.

С целью более глубокого понимания смысла использованных функций, читателю рекомендуется обратиться к документации OpenGL [11].

Для создания шейдерной программы предлагается использовать функцию следующей сигнатуры:

```
GLuint createProgram(GLuint vsh, GLuint fsh)
```

– `vsh` – дескриптор вершинного шейдера;
– `type` – дескриптор фрагментного шейдера;
– возвращаемое значение – ненулевой дескриптор программы, если создание прошло успешно; 0, если при создании произошла ошибка.

Тело функции приведено в листинге 2.

Листинг 2. Функция создания шейдерной программы

```
GLuint createProgram(GLuint vsh, GLuint fsh)
{
    GLuint result = glCreateProgram();

    glAttachShader(result, vsh);
    glAttachShader(result, fsh);

    glLinkProgram(result);

    GLint linked;
    glGetProgramiv(result, GL_LINK_STATUS,
                   &linked);

    if (!linked)
    {
        GLint infoLen = 0;
        glGetProgramiv(result, GL_INFO_LOG_LENGTH,
                       &infoLen);
        if (infoLen > 0)
        {
            char infoLog[infoLen];
            glGetProgramInfoLog(result, infoLen,
                                NULL, infoLog);
            cout << "Shader program linking error"
                 << endl << infoLog << endl;
        }
        glDeleteProgram(result);
        return 0;
    }

    return result;
}
```

В строке (3) происходит создание объекта шейдерной программы. Затем, в строках (5–6) к этому объекту присоединяются шейдеры. В строке (8) происходит линковка программы, а в строке (11) – запрос статуса компоновщика. Строки (16–28), аналогично предыдущей функции, посвящены выводу на консоль описания потенциальных ошибок. В случае же успешной линковки функция возвращает дескриптор созданной и готовой к использованию шейдерной программы.

Важно отметить, что после сборки программы сами шейдерные объекты, участвовавшие в её создании, оказываются более не нужны, и их можно удалить функцией `glDeleteShader`. Шейдерная программа, в свою очередь, должна существовать до тех пор, пока сцене нужны те эффекты, которые она реализует. После этого она также должна быть удалена функцией `glDeleteProgram`.

В данном примере, чтобы не усложнять структуру тестового приложения, дескриптор шейдерной программы имеет смысл объявить как глобальную переменную, создать программу в функции `init` и удалить в функции `cleanup`.

Строковые константы кодов вершинного и фрагментного шейдера приведены в листингах 3 и 4 соответственно.

Листинг 3. Код вершинного шейдера

```
1  const GLchar vsh[] =
2  "#version 330\n"
3  ""
4  "layout(location = 0) in vec2 a_position;"
5  "layout(location = 1) in vec3 a_color;"
6  ""
7  "out vec3 v_color;"
8  ""
9  "void main()"
10 "{"
11 "    v_color = a_color;"
12 "    gl_Position = vec4(a_position, 0.0, 1.0);"
13 "}";
```

Листинг 4. Код фрагментного шейдера

```
1  const GLchar fsh[] =
2  "#version 330\n"
3  ""
4  "in vec3 v_color;"
5  ""
6  "layout(location = 0) out vec4 o_color;"
7  ""
8  "void main()"
9  "{"
10 "   o_color = vec4(v_color, 1.0);"
11 "};
```

В первой строке указывается директива компилятору касательно используемой версии GLSL. Поскольку, как уже отмечалось выше, этот язык по мере своего развития претерпевал ощутимые изменения, указывать версию необходимо в явном виде. Версия 330 соответствует стандарту OpenGL 3.3.

Для упрощения дескрипторы атрибутов заданы в коде шейдера в явном виде. Вершинный шейдер имеет два атрибута – `vec2 a_position` (пространственные координаты) и `vec3 a_color` (цвет), т. е. ожидает данные в формате (x, y, r, g, b). Кроме того, он готовит данные для фрагментного шейдера: `vec3 v_color`. В своей функции `main` он передаёт данные из атрибута цвета в соответствующий вэриинг, а также означивает позицию текущей вершины, собирая четырёхкомпонентный вектор однородных координат из принятых на вход пространственных данных. Как можно видеть, здесь не используется трансформация вершин: шейдер ожидает, что координаты выводимого объекта уже приведены к NDC.

Фрагментный шейдер определяет входной канал `vec3 v_color`, соответствующий выходному каналу вершинного шейдера, а также выходной канал `vec4 o_color` для нулевой цели рендеринга (буфера цвета). Единственное, что происходит в его функции `main` – означивание выходного канала пришедшими

с предыдущих ступеней конвейера данными, дополняя их до четырёхкомпонентного вектора (добавляя альфа-канал).

Задание: Добавить в тестовое приложение создание шейдерной программы при помощи функций, приведённых в данной главе.
Подсказка: Обратить внимание на корректную и своевременную очистку всех созданных объектов.

7. Представление объектов сцены

Объекты сцены в компьютерной графике вообще и в OpenGL в частности представляются совокупностью *геометрической модели* (англ. *Model*) и *материала* (англ. *Material*). Геометрическая модель – это совокупность свойств формы объекта. Материал – это совокупность визуальных свойств объекта. Обе составляющих объекта сцены определяются связанными с объектом данными и конкретной программой графического конвейера (шейдерами).

7.1. Полигональная сетка

Данные объекта могут храниться в виде достаточно сложной структуры, включая в себя описание его вершин и связей между ними, описание оптических свойств его поверхности и т. д. Вершины и их связи называются *полигональной сеткой объекта* (англ. *Mesh*).

В самом простом случае эта сетка представлена одним массивом, в котором вершины последовательно объединяются в примитивы. Однако, если представить составленный из двух треугольников четырёхугольник, становится очевиден недостаток такого подхода: идущие строго последовательно вершины, описывающие некоторую сетку, с высокой вероятностью приведут к дублированию данных. На сложных моделях такое дублирование может привести к серьёзным накладным расходам.

Эта проблема решается введением индексации: отдельного массива, элементами которого являются номера вершин. Этот массив задаёт порядок следования вершин и позволяет переиспользовать данные, хранящиеся однократно.

В более сложном случае один объект может быть представлен несколькими массивами, хранящими разные атрибуты его вершин (при этом все они отправляются на конвейер одновременно за один вызов отрисовки), или же несколькими принципиально различными полигональными сетками (которые отправляются на конвейер последовательно, разными вызовами отрисовки). Это зависит от

уровня абстракции, который мы используем, говоря об «объекте сцены». Если, например, объектом сцены является модель автомобиля (обладающая при этом достаточно высокой детализацией), в её состав, скорее всего, войдёт несколько полигональных сеток, представляющих поверхности принципиально разных свойств: металлический корпус, стёкла, светящиеся фары, резиновые колёса и т. д. Для высококачественной визуализации такой модели потребуется осуществить несколько вызовов отрисовки, используя при этом различные шейдеры для различных «деталей».

Однако в графических приложениях, работающих в реальном времени, как правило, используется диаметрально противоположный подход: объединение возможно большего числа объектов в один массив, с целью минимизации числа вызовов отрисовки. Отправка большой порции данных на конвейер за один раз оказывается значительно эффективнее, чем отправка нескольких более мелких пакетов. Более того, иметь один массив вершин, собирающий в себе все используемые атрибуты, эффективнее, чем несколько массивов, по одному атрибуту в каждом.

В данном пособии мы рассмотрим работу лишь с одним массивом, но обобщение до работы с несколькими является тривиальным: все шаги, необходимые для отправки на конвейер одного массива, надо последовательно проделать со всеми имеющимися, а затем вызвать отрисовку.

7.2. Хранение объектов сцены в OpenGL

Для хранения данных объекта сцены в OpenGL 3.3+ имеется три структуры данных:

1. *Буфер вершин* (англ. *Vertex Buffer Object*) – структура для хранения массива вершин, представленных их атрибутами. Как правило, элементы этого массива имеют тип `float`.
2. *Буфер индексов* (англ. *Index Buffer Object*) – структура для хранения массива индексов, задающих порядок обхода вершин для объединения их в примитивы. Как правило, элементы этого массива имеют тип `unsigned int` (на настольных ком-

пьютерах, следовательно, адресовать можно 2^{32} вершин) или `unsigned short` (на мобильных устройствах, следовательно, адресовать можно 2^{16} вершин).

3. *Буфер массива вершин* (англ. *Vertex Array Object*) – структура для хранения отображения массива вершин на входы шейдера.

Все три структуры, как это принято в OpenGL, могут создаваться, настраиваться и уничтожаться при помощи специальных API функций. При этом фактические данные хранятся в видеопамяти, а в основном приложении доступны лишь соответствующие целочисленные дескрипторы.

7.3. Реализация

Для удобства работы предлагается хранить дескрипторы всех связанных с объектом сцены буферов в отдельном классе:

```
class Model
{
public:
    GLuint vbo;
    GLuint ibo;
    GLuint vao;
};
```

Объект этого класса предлагается объявить в виде глобальной переменной (как и дескриптор шейдерной программы).

Большинство графических движков используют объектно-ориентированную парадигму для создания обёртки, хранящей внутри себя дескрипторы низкоуровневых структур данных и методы управления этими структурами. При желании читатель может самостоятельно создать собственный легковесный графический движок, однако мы не будем углубляться в проектирование архитектуры подобного рода программного обеспечения и ограничимся лишь самой простой группировкой внутри одного класса дескрипторов, описывающих различные низкоуровневые составляющие одной и той же высокоуровневой сущности (3D-модели).

В качестве первого опыта визуализации выведем на экран самый тривиальный объект – треугольник. В предыдущей главе были подготовлены шейдеры, ожидающие данные в формате (x, y, r, g, b), причём координаты (x, y) должны быть указаны в NDC. Внешний вид тестового треугольника приведён на рис. 12.

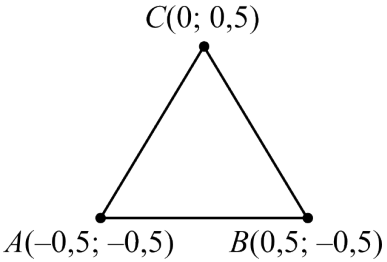

Рис. 12. Треугольник, подлежащий выводу в качестве первого опыта визуализации

Несмотря на то что треугольник в данном случае будет всего один, воспользуемся индексацией, чтобы сразу создать расширяемую программу.

Следует отметить, что при визуализации имеет значение порядок обхода вершин. Дело в том, что системе могут быть заданы разные настройки для вывода *передних* (англ. *Front Face*) и *задних* (англ. *Back Face*) граней. Передней по умолчанию считается грань, вершины которой после проекции на экран идут *против часовой стрелки* (англ. *Counterclockwise*, CCW), а задней – *по часовой стрелке* (англ. *Clockwise*, CW). При этом существует настройка, которая меняет правило определения на обратное.

Определение происходит на уровне каждого отдельного треугольника аппаратно, путём нахождения векторного произведения векторов, построенных на его сторонах в порядке обхода вершин, и проверки знака координаты z (глубины) полученного результата.

Для создания модели предлагается следующая сигнатура функции:

```
Model createModel()
```

Эта функция, как можно догадаться по отсутствию параметров, не будет такой же универсальной, как функции создания шейдеров и шейдерных программ, поскольку универсальность в данном случае означала бы некоторое усложнение кода. Цель этой функции – на простом примере показать основной механизм подготовки данных объекта сцены. Тело функции приведено в листинге 5.

Листинг 5. Функция создания простого объекта сцены

```
Model createModel()
{
    const GLfloat vertices[] =
    {
        -0.5f, -0.5f, 1.0f, 0.0f, 0.0f,
         0.5f, -0.5f, 0.0f, 1.0f, 0.0f,
         0.0f,  0.5f, 1.0f, 1.0f, 0.0f,
    };

    const GLuint indices[] =
    {
        0, 1, 2,
    };

    Model result;

    glGenVertexArrays(1, &result.vao);
    glBindVertexArray(result.vao);

    glGenBuffers(1, &result.vbo);
    glBindBuffer(GL_ARRAY_BUFFER,
                 result.vbo);
    glBufferData(GL_ARRAY_BUFFER,
                 15 * sizeof(GLfloat),
                 vertices,
                 GL_STATIC_DRAW);

    glGenBuffers(1, &result.ibo);
```

```
29      glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
30                   result.ibo);
31      glBufferData(GL_ELEMENT_ARRAY_BUFFER,
32                   3 * sizeof(GLuint),
33                   indices,
34                   GL_STATIC_DRAW);
35
36      glEnableVertexAttribArray(0);
37      glVertexAttribPointer(0, 2, GL_FLOAT,GL_FALSE,
38                            5 * sizeof(GLfloat),
39                            (const GLvoid *)0);
40      glEnableVertexAttribArray(1);
41      glVertexAttribPointer(1, 3, GL_FLOAT,GL_FALSE,
42                            5 * sizeof(GLfloat),
43                            (const GLvoid *)
44                            (2 * sizeof(GLfloat)));
45
46      return result;
47  }
```

В строке (3) создаётся массив вершин объекта по оговоренному ранее формату (5 значений типа `float` на вершину). Как правило, массивы вершин загружаются из файлов (куда их записывают заранее с помощью визуальных редакторов 3D-графики), однако *процедурные* (программно сгенерированные) модели также не являются редкостью.

В строке (10) создаётся массив индексов, который в данном примере обеспечивает тривиальный обход элементов массива вершин подряд.

В строке (17) создаётся буфер массива вершин. Помимо слова `Create`, OpenGL использует ещё и слово `Gen` (от англ. *Generate* – генерировать) в аналогичном значении: чтобы обозначить функцию, создающую новый объект и передающую ответственность за него вызывающему. Отличием является то, что функции со словом `Gen` за один вызов могут создать не один, а произвольное число объектов (в пределах максимально возможного числа объектов данного типа,

которое определяется аппаратными возможностями конкретной системы). Однако в рассматриваемом примере буфер массива вершин создаётся всего один.

В строке (18) созданный буфер активируется, и все последующие настройки составляющих трёхмерного объекта будут автоматически сохранены в нём. В дальнейшем можно будет лишь повторно активировать его, чтобы мгновенно (вызовом лишь одной функции активации) восстановить состояние графического конвейера, необходимое для вывода соответствующего трёхмерного объекта. Такой подход «пакетного» хранения и применения настроек способен значительно увеличить производительность, а потому стал обязательным к использованию в OpenGL 3.3+ (в более ранних версиях он был представлен лишь опциональным расширением, а в обычном режиме приходилось перед каждой отрисовкой очередного объекта применять все необходимые настройки заново вручную).

В строке (20) создаётся и в строке (21) активируется буфер вершин. Первый параметр функции активации `glBindBuffer` определяет тип буфера – вершинный (константа `GL_ARRAY_BUFFER`) или индексный (константа `GL_ELEMENT_ARRAY_BUFFER`). Из такого кода следует, что типизация буфера происходит динамически и не сохраняется в буфере. Однако фактически семантика использования буфера должна быть сохранена в использующем его приложении, поскольку, например, интерпретация вершинного буфера как индексного неминуемо приведёт к ошибкам работы конвейера из-за некорректных обращений к памяти.

В строке (23) происходит копирование данных из массива вершин в активный буфер вершин. В один момент времени активным может быть только один буфер конкретного типа, однако буферы разных типов могут быть активированы одновременно. Поэтому функция заполнения `glBufferData` принимает в качестве первого параметра тип буфера. Второй параметр этой функции – размер копируемых данных *в байтах*. Третий параметр – нетипизированный указатель на массив данных. Четвёртый параметр – константа, определяющая режим использования буфера, а именно – является ли он статичным (неизменным) или его содержимое будет меняться в процессе

работы приложения. Такая информация позволяет системе оптимизировать доступ к буферу. В приведённом примере буфер объявлен статичным.

Данные буфера хранятся в видеопамяти, поэтому вызов `glBufferData` оказывается достаточно дорогостоящим, осуществляя копирование через шину. В связи с этим заполнение буферов рекомендуется осуществлять на этапе подготовки сцены до начала циклической визуализации.

Далее в строках (28–34) совершаются действия по созданию буфера индексов. Эти действия аналогичны предыдущим, отличие составляет только тип буфера.

После этого настраивается состояние, т. е. связь атрибутов вершин со входами вершинного шейдера.

Сначала в строке (36) разрешается использование массива данных для атрибута с дескриптором 0. Этому атрибуту соответствует `a_vertex` из шейдера, описанного в предыдущей главе (дескриптор был определён для этого атрибута в явном виде при помощи спецификатора `layout`). Следует отметить, что в отличие от шейдеров, шейдерных программ и буферов, дескриптор 0 является валидным для атрибутов.

Затем функцией `glVertexAttribPointer` устанавливается относительный указатель на начало данных атрибута в буфере и шаг выборки атрибутов. Сигнатура этой функции довольно сложна и часто является источником ошибок, поэтому разберём её подробно:

```
void glVertexAttribPointer(GLuint index,
                           GLint size,
                           GLenum type,
                           GLboolean normalized,
                           GLsizei stride,
                           const GLvoid *pointer);
```

– `index` – дескриптор атрибута;
– `size` – число компонентов атрибута (число элементов того типа, в котором хранятся данные в буфере вершин, связанных с одним атрибутом);

– `type` – константа, определяющая тип данных, хранящихся в буфере (чаще всего атрибуты вершин представляют собой значения типа `float`, чему соответствует константа `GL_FLOAT`);
– `normalized` – флаг, определяющий, должны ли значения атрибута быть нормированы, т. е. приведены к отрезку $[-1; 1]$ для случая знаковых и $[0; 1]$ для случая беззнаковых чисел (этот флаг используется в том случае, если значения атрибутов хранятся в виде целых чисел);
– `stride` – шаг *в байтах*, по которому нужно выбирать данные атрибутов каждой следующей вершины (чаще всего это значение вычисляется как произведение числа компонентов вершины и размера, обусловленного типом этих компонентов);
– `pointer` – смещение *в байтах* относительно начала буфера, по которому располагается первый компонент атрибута, приведённое к указателю.

В строках (40–44) аналогично настраивается атрибут с дескриптором 1, которому соответствует заданный в шейдере `a_color`.

В результате модель, представленная тремя буферами, оказывается полностью настроенной и готовой к визуализации.

Следует обратить внимание, что использованные буферы, как и шейдерную программу, необходимо удалить после завершения цикла визуализации сцены (в функции `cleanup`). Для удаления буферов вершин и индексов используется функция `glDeleteBuffers`, для удаления буфера массива вершин – функция `glDeleteVertexArrays`. Обе функции принимают на вход массив дескрипторов и их число.

Задание: Добавить в тестовое приложение создание модели при помощи функций, приведённых в данной главе.

Подсказка: Обратить внимание на корректную и своевременную очистку всех созданных объектов.

8. Визуализация сцены

Использованные в тестовом приложении библиотеки GLFW и GLEW при своей инициализации создают графический контекст и делают графический конвейер доступным для дальнейшей работы. Однако чтобы осуществить визуализацию, необходимо явным образом указать некоторые настройки.

8.1. Порт просмотра

Как уже отмечалось ранее, для организации проекции объектов на экран необходимо, с одной стороны, создать преобразование координат объектов сцены в NDC, а с другой – задать порт просмотра. Преобразование координат лежит на программисте и осуществляется при помощи матриц. Характеристики порта просмотра, напротив, являются частью машины состояний OpenGL и преобразование порта просмотра система выполняет автоматически.

Преобразование порта просмотра схематично изображено на рис. 13.

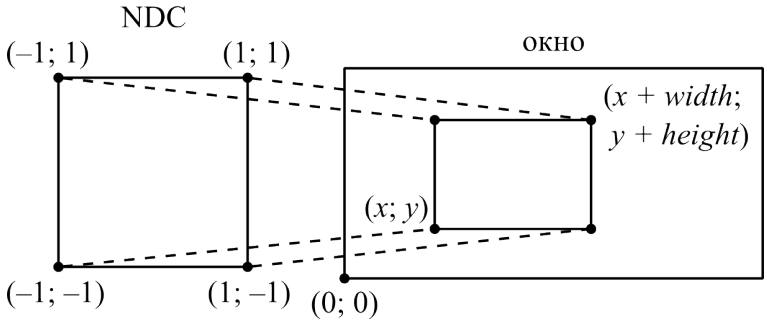

Рис. 13. Преобразование порта просмотра в OpenGL

Как видно из рис. 13, характеристикой порта просмотра являются координаты его левого нижнего угла в системе координат окна

(начало которой находится в левом нижнем углу окна), а также ширина и высота. За единицу в системе координат окна согласно спецификации OpenGL принят *физический* пиксель экрана. Как уже отмечалось ранее, в некоторых графических интерфейсах физические пиксели экрана отличаются от *логических*, в которых задаются координаты элементов (например, на ретина-дисплеях). Однако порт просмотра, как и все остальные сущности, описанные спецификацией OpenGL, относятся к низкоуровневой графике и оперируют физическими пикселями.

Для задания характеристики порта просмотра используется функция `glViewport`. Чтобы осуществить корректную настройку конвейера, вызов этой функции необходимо добавить в функцию `reshape`, которая вызывается каждый раз, когда изменяется размер окна. Код, обеспечивающий совпадение порта просмотра с окном, приведён в листинге 6.

Листинг 6. Функция обратного вызова на изменение размера окна

```
1 void reshape(GLFWwindow *window,
2               int width, int height)
3 {
4     glViewport(0, 0, width, height);
5 }
```

8.2. Вызов отрисовки

Для осуществления визуализации объекта сцены необходимо:

1. Активировать соответствующую объекту шейдерную программу (если она не активна).
2. Активировать соответствующий объекту буфер массива вершин модели (если он не активен).
3. Выполнить вызов отрисовки.

Обычно сцены состоят из большого числа разнообразных объектов, некоторые из которых могут использовать одни и те же шейдерные программы или модели (отличаясь, например, положением и

значениями каких-либо юниформ-переменных). В этом случае рекомендуется группировать объекты с общими ресурсами и выводить их друг за другом, минимизируя изменения состояния конвейера: переключения настроек тоже требуют времени, хотя это далеко не такие дорогостоящие операции, как копирование данных из оперативной в видеопамять.

В нашем случае сцена состоит лишь из одного объекта. Код функции отрисовки, циклически вызываемой во время выполнения приложения, приведён в листинге 7.

Листинг 7. Функция отрисовки сцены

```
1  void draw()
2  {
3      glClear(GL_COLOR_BUFFER_BIT);
4
5      glUseProgram(g_shaderProgram);
6      glBindVertexArray(g_model.vao);
7
8      glDrawElements(GL_TRIANGLES,3,GL_UNSIGNED_INT,
9                     (const GLvoid *)0);
10 }
```

Предполагается, что `g_shaderProgram` – глобальная переменная типа `GLuint`, хранящая дескриптор шейдерной программы, а `g_model` – глобальная переменная типа `Model`, хранящая буферы модели.

В строке (3) осуществляется очистка буфера цвета. В данном случае очистку можно и не производить, так как сцена не изменяется с течением времени. Однако в общем случае этот вызов необходим для удаления данных, сгенерированных на предыдущем кадре.

В строках (5–6) осуществляется активация шейдерной программы и буфера массива вершин модели. В данном случае эту активацию можно было бы произвести однократно, однако в общем случае, когда сцена представлена большим числом объектов, выводящихся при помощи разных шейдеров, требуется каждый раз заново перенастраивать состояние конвейера.

Следует отметить, что хотя для отрисовки используется лишь буфер массива вершин, буферы вершин и индексов должны присутствовать в памяти до тех пор, пока требуется визуализировать соответствующую им модель. Буфер массива вершин является лишь своего рода дескриптором трёхмерного объекта и хранит в себе лишь настройки, но не сами данные.

В строке (8) происходит вызов отрисовки, обеспечивающий отправку данных на графический конвейер.

Вызов отрисовки представлен в OpenGL двумя альтернативными командами: `glDrawArrays` и `glDrawElements`. Первая из них игнорирует буфер индексов (и допускает полное его отсутствие), совершая обход вершин в порядке их следования в вершинном буфере. Вторая использует индексацию, заданную либо индексным буфером, либо, если буфер индексов не активирован, массивом из оперативной памяти (указатель на который передаётся одним из параметров функции). Однако использование для индексации массивов из оперативной памяти неэффективно, так как приводит к копированию данных в видеопамять непосредственно перед отрисовкой (а не заранее, как в случае с индексным буфером). Другое назначение указателя (в случае, если активирован буфер индексов) – указание смещения относительно начала индексного буфера, чтобы была возможность вывести лишь часть модели.

Обе функции принимают в качестве параметров константу, определяющую тип выводимых примитивов и число вершин для обработки. Таким образом, одну и ту же модель можно выводить разными способами, в частности – фрагментарно.

Поскольку модель в данном случае использует индексацию, отправка данных на конвейер осуществляется функцией `glDrawElements`. Рассмотрим её сигнатуру:

```
void glDrawElements(GLenum mode,
                    GLsizei count,
                    GLenum type,
                    const GLvoid *indices);
```

– `mode` – константа, определяющая тип примитива (с доступными типами примитивов читателю предлагается познакомиться са-

мостоятельно, воспользовавшись официальной документацией OpenGL [11]);
- `count` – число вершин для обработки;
- `type` – тип индексов (для настольных компьютеров обычно используется `GL_UNSIGNED_INT`, для мобильных устройств на данный момент поддерживается лишь `GL_UNSIGNED_SHORT`);
- `indices` – указатель на массив индексов в оперативной памяти (если буфер индексов не активирован), либо смещение *в байтах* для буфера индексов (если он активирован), приведённое к указателю.

Сигнатуру функции `glDrawArrays` читателю предлагается изучить самостоятельно в документации OpenGL [11].

В результате тестовое приложение должно вывести на экран треугольник, закрашенный цветовым градиентом (градиент возникает за счёт интерполяции цвета, заданного в вершинах).

Задание: Добавить в тестовое приложение указанные в данной главе вызовы функций и добиться отображения закрашенного градиентом треугольника. Временно исключить буфер индексов и перевести отрисовку на метод `glDrawArrays`.

8.3. Разогрев конвейера

Следует отметить одну неочевидную особенность графического конвейера – необходимость «разогрева».

Проблема в том, что первый вызов отрисовки с каждым конкретным сочетанием шейдерной программы и настроек отображения модели на входы этой программы выполняется дольше, чем последующие вызовы. Это связано с валидацией состояния, которую производит графический драйвер. Результат валидации кешируется, поэтому повторные вызовы отрисовки не требуют дополнительных проверок и выполняются быстрее.

В тестовом приложении используются слишком простые шейдеры, поэтому разница между первым и последующими вызовами

отрисовки совершенно незаметна. Однако в более сложных проектах разница в скорости может достигать *десятков раз*, в силу чего первый кадр визуализации сцены готовится ощутимо дольше, чем последующие.

Хуже того, поскольку валидация проводится каждый раз, когда конвейер «сталкивается» с необходимостью выполнить новую шейдерную программу или принять новую конфигурацию данных (новый вид отображения атрибутов вершин модели на входы шейдера), увеличение временных затрат на кадр может возникнуть посреди работы графической программы, когда на сцене в соответствии с логикой приложения появляется объект с новыми визуальными свойствами. Для пользователя такая ситуация выглядит как «внезапный лаг», нарушение плавности движений на сцене и т. д., что сильно снижает эргономику использования приложения.

Чтобы избежать такой ситуации, перед началом визуализации сцены необходимо «разогреть» конвейер путём «черновой» визуализации всех объектов, которые когда-либо будут появляться на сцене в процессе работы графического приложения. В принципе, не требуется визуализировать сами модели объектов: достаточно лишь воссоздать конфигурацию данных каждого объекта (массив вершин при этом можно использовать упрощённый, например, содержащий лишь один примитив).

Результаты «разогревающей» конвейер визуализации не нужно показывать на экране: сразу после всех «черновых» вызовов отрисовки можно почистить буфер цвета и запускать основной цикл графического приложения.

Однако перед «черновой» визуализацией на некоторых программно-аппаратных платформах (фактическое поведение зависит от аппаратуры) требуется отобразить *минимум два кадра* с некоторым содержательным изображением, чтобы заполнить страницы видеопамяти двойного буфера (при использовании двойной буферизации). Как правило, в качестве такого «разогревающего» изображения используют некоторую *заставку* (англ. *Splash Screen*), показываемую в момент старта графического приложения.

9. Процедурная закраска

Аппарат шейдеров даёт возможность организовывать сложные визуальные эффекты при отображении объектов. Типичным примером является *процедурная закраска* (*англ. Procedural Shading*) – вычисление цвета фрагментов полигонов по некоторому алгоритму, отличному от тривиальной интерполяции. В процедурной закраске, как правило, участвуют атрибуты вершин, напрямую не выражающие цвет (как в предыдущем примере), а являющиеся параметрами алгоритма раскрашивания.

Предлагается доработать тестовое приложение, реализовав в нём процедурную закраску. Сначала предлагается добавить к имеющемуся в приложении треугольнику ещё один так, чтобы получился прямоугольник (на прямоугольнике будет удобнее проводить эксперименты с процедурной закраской). При этом предлагается избавиться от атрибутов, задающих цвет вершин, а вместо них использовать один дополнительный атрибут $t \in [0; 1]$. Таким образом, каждая вершина будет характеризоваться тройкой чисел (x, y, t).

Вид прямоугольника со значениями атрибутов вершин представлен на рис. 14.

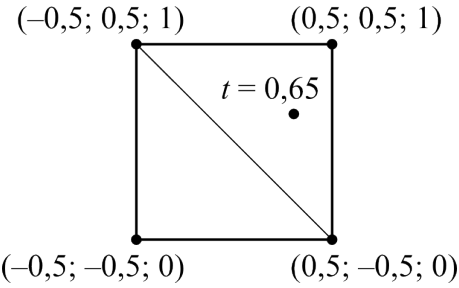

Рис. 14. Прямоугольник для тестирования процедурной закраски

Для примера на рис. 14 указана некоторая внутренняя точка со значением атрибута t в ней. Этот атрибут будет в данном случае являться параметром процедурной закраски.

Далее предлагается изменить код шейдеров на приведённый в листинге 8.

Листинг 8. Код шейдеров процедурной закраски

```
const GLchar vsh[] =
"#version 330\n"
""
"layout(location = 0) in vec2 a_position;"
"layout(location = 1) in float a_t;"
""
"out float v_t;"
""
"void main()"
"{"
"    v_t = a_t;"
"    gl_Position = vec4(a_position, 0.0, 1.0);"
"};

const GLchar fsh[] =
"#version 330\n"
""
"in float v_t;"
""
"layout(location = 0) out vec4 o_color;"
""
"void main()"
"{"
"   o_color = vec4(sin(v_t * 62.83),0.0,0.0,1.0);"
"};
```

Результат работы приложения показан на рис. 15.

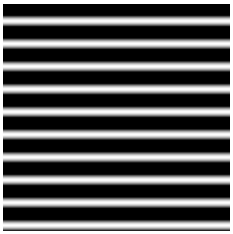

Рис. 15. Результат процедурной закраски по алгоритму из листинга 8 (цвета изменены на более контрастные в типографских целях)

Как можно видеть на рис. 15, на экран выводится прямоугольник, закрашенный 10 горизонтальными линиями. За их формирование отвечает вектор цвета `vec4(sin(v_t * 62.83), 0.0, 0.0, 1.0)` из листинга 8. Параметр `v_t` изменяется от 0 до 1, в результате чего синус проходит примерно 10 периодов, попеременно возвращая значения в отрезке $[-1; 1]$. При интерпретации цвета отрицательные значения зануляются, что приводит к появлению чёрного. Положительные же значения приводят к постепенному нарастанию, а затем – затуханию интенсивности красного.

Немного усложнив алгоритм закраски, можно получить изображения, подобные продемонстрированным на рис. 16.

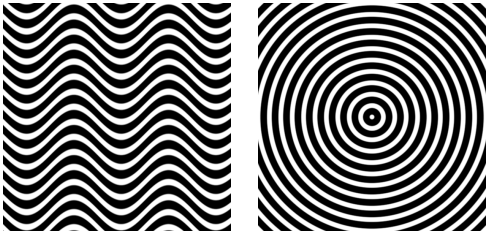

Рис. 16. Результат процедурной закраски волнистыми линиями (слева) и концентрическими окружностями (справа)

Задание: Изменить исходный код шейдеров, массивы вершин и индексов, настройку отображения атрибутов на входы вершинного шейдера и вызов отрисовки так, чтобы на экран был выведен прямоугольник, закрашенный линиями, как на рис. 15. Изменить приложение так, чтобы на экран был выведен прямоугольник, закрашенный волнистыми линиями, как на рис. 16 слева. Изменить приложение так, чтобы на экран был выведен прямоугольник, закрашенный концентрическими окружностями, как на рис. 16 справа.

Подсказка: Для достижения закраски волнистыми линиями и окружностями следует добавить ещё один атрибут вершин p, «направив» его изменение вдоль оси X. Функция изменения яркости будет иметь вид $f = f(t, p)$. Яркость в данном случае аналогична высоте поверхности $y = f(x, y)$. В принципе, в конкретно данном случае в качестве параметров для вычисления яркости можно использовать координаты вершин, полностью избавившись от дополнительных атрибутов. Однако этого делать не рекомендуется, поскольку в реальных ситуациях, при закраске более сложных моделей, завязка на координаты может нарушить единообразие обработки всех полигонов объекта сцены.

10. Обработка объёмных структур

В предыдущих главах все примеры визуализации ограничивались двумерными объектами, координаты вершин которых дополнялись значениями $z = 0$, $w = 1$ лишь в вершинном шейдере. Более того, все координаты в основной программе задавались в NDC и не использовались никакие преобразования размещения и проекции. Таким образом, работа с трёхмерной графикой велась в неявном виде: с точки зрения пользователя сцена была плоской. Однако OpenGL (и подобные ему стандарты) в первую очередь решает задачи обработки объёмных геометрических структур.

Переход в трёхмерное пространство, однако, влечёт за собой, помимо дополнительных степеней свободы размещения объектов, также и дополнительные алгоритмические сложности.

10.1. Буфер глубины

Важным вопросом при визуализации объектов сцены является определение и корректная обработка перекрытий в случае, если проекции объектов имеют пересечение на плоскости экрана.

В двумерной графике (когда каждый объект характеризуется лишь двумя координатами x и y), как правило, перекрытия полностью зависят от порядка поступления данных на конвейер (по сути, используется дисциплина FIFO): объект, который был отправлен на конвейер раньше, оказывается перекрыт объектом, отправленным позже. Такой подход достаточно удобен, так как позволяет программисту разграничивать «передний» и «задний» планы путём сортировки списка объектов. Порядок вывода в контексте двумерной графики часто называют *z-порядком* (англ. *Z-Order*), обозначая тем самым, что сцена имеет «псевдоглубину».

В трёхмерной графике (в случае когда объекты характеризуются тремя координатами x, y и z) такой подход не применим, так как z-порядок здесь явным образом определяется самими объектами и для корректной визуализации сцены не должен зависеть от очерёд-

ности их обработки. Например, если более близкий к наблюдателю объект был выведен раньше, он не должен быть перекрыт более дальним, который был отправлен на конвейер позднее.

При этом сортировка объектов и даже отдельных граней объектов по удалённости в общем случае не обеспечит корректного отображения, поскольку объекты могут быть невыпуклыми и могут пересекаться в пространстве. На рис. 17 продемонстрирована ситуация, когда любой порядок вывода привёл бы к неверному результату. Кроме того, ввиду потенциально изменяющегося ракурса и взаимного расположения объектов, сортировку необходимо было бы осуществлять после каждого изменения, что в общем случае неэффективно (в особенности, если речь идёт о гранях объектов, поскольку изменение порядка их следования означает как минимум изменение массива индексов).

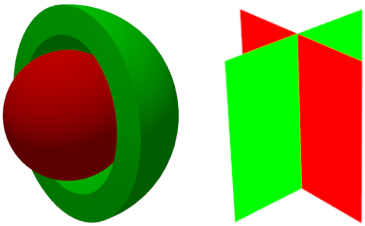

Рис. 17. Примеры объектов, корректное перекрытие проекций которых не обеспечить порядком отрисовки

Для обеспечения корректных перекрытий, не зависящих от порядка вывода объектов и определяющихся их реальным взаимным расположением в трёхмерном пространстве, используют специальный механизм, называемый *тестом глубины* (англ. *Depth Test*). Этот механизм базируется на структуре данных, называемой *буфером глубины* (англ. *Depth Buffer*), или *z-буфером* (англ. *Z-Buffer*).

Буфер глубины является частью буфера кадра и представляет собой двумерную матрицу, по размерности совпадающую с буфером цвета. Как правило, память под него выделяют вместе с буфером цвета при создании буфера кадра в графическом контексте. Ячейки бу-

фера цвета могут иметь разную разрядность, но предназначены для хранения единственного числового значения (а не вектора, как в случае буфера цвета). На сегодняшний день разрядность буфера глубины составляет, как правило, 16 или 24 бита.

Концептуально алгоритм теста глубины состоит в следующем (предположим, что ось глубины направлена на наблюдателя):

1. Перед началом визуализации кадра буфер глубины заполняется некоторыми начальными значениями, чаще всего – нулями.
2. На этапе растеризации для каждого фрагмента, соответствующего обрабатываемому примитиву, методом билинейной интерполяции (который был рассмотрен ранее в контексте интерполяции атрибутов вершин) определяется значение глубины (координата z).
3. Значение глубины z сравнивается с содержимым z_b соответствующей ячейки буфера глубины.
   3.1. Если $z \geq z_b$, данные фрагмента записываются в ячейки соответствующих ему целей рендеринга (например, цвет в ячейку буфера цвета), а z записывается в ячейку буфера глубины.
   3.2. Если $z < z_b$, фрагмент отбрасывается (относительно него более не происходит никаких действий, а графический процессор переходит к обработке следующего).

Таким образом, более близкие к наблюдателю фрагменты маскируют более дальние, не давая им выводиться. В свою очередь, более дальние фрагменты перезаписываются более близкими. За счёт этого обеспечиваются правильные перекрытия.

Следует, однако, обратить внимание, что разрядность буфера глубины невелика (она продиктована аппаратными ограничениями видеокарт). Из-за этого при сохранении значений в его ячейки происходит потеря точности, которая в последствии может привести к неверным результатам теста глубины, в особенности по отношению к копланарным полигонам.

Ошибки теста глубины называют *борьбой глубин* (англ. *Z-Fighting*). Пример такой ситуации показан на рис. 18.

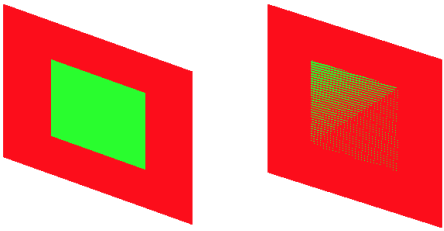

Рис. 18. Корректное отображение полигонов (слева) и ситуация борьбы глубин (справа)

Дефекты, возникающие из-за борьбы глубин, резко снижают качество изображения, а на динамически меняющихся сценах создают «мелькание», поскольку тест глубины оказывается неустойчив и при небольших изменениях ракурса даёт резкое изменение результата визуализации.

Для избежания борьбы глубин категорически не рекомендуется создавать модели с копланарными пересекающимися полигонами.

Как правило, графические API дают возможность настраивать функции сравнения и записи, отмеченные в шаге (3.1) алгоритма теста глубины. То есть настройками состояния конвейера можно, например, инвертировать работу буфера глубины для решения каких-либо специфических задач, а также задать функцию модификации значения z перед записью в его ячейку.

Так, например, фрагментный шейдер может вычислять, по сути, произвольное значение глубины фрагмента, которое затем будет использовано для теста глубины, и при прохождении теста записано в буфер. Такая ситуация, однако, в ряде случаев может снизить производительность работы: многие графические процессоры в целях оптимизации обработки сцены производят т. н. *ранний тест глубины* (англ. *Early Depth Test*). Он производится *до* запуска фрагментного шейдера, если этот шейдер не изменяет значение глубины, что позволяет сэкономить время вычисления цвета фрагмента, который не вносит вклад в итоговое изображение (если в буфере цвета на этом месте уже есть «более близкий к наблюдателю» фрагмент).

Определение возможности раннего теста глубины осуществляется перед началом прогона данных по конвейеру путём статического анализа кода фрагментного шейдера.

Для частичного решения проблемы потери точности в буфере глубины значения в нём хранятся не в прямом, а в преобразованном виде, причём преобразование нелинейно. Чаще всего используется следующая формула:

$$z_b = \left[\frac{2^n}{z_{far} - z_{near}} \left(z_{far} + \frac{z_{far}\, z_{near}}{z} \right) \right],$$

где z_b – целочисленное значение, сохраняемое в буфер глубины,
n – разрядность буфера глубины,
z_{far} – расстояние от наблюдателя до дальней плоскости отсечения,
z_{near} – расстояние от наблюдателя до ближней плоскости отсечения,
z – значение глубины фрагмента.

Такой способ хранения обеспечивает достаточно высокую точность представления глубин около ближней плоскости отсечения и уменьшение точности по мере удаления. Это оправданно, поскольку ближние к наблюдателю объекты занимают на экране больше места, а потому обладают большей детализацией и требуют более точной обработки (в частности, в отношении мелких деталей, расположенных по оси глубины близко друг к другу и рискующих вступить в «борьбу»).

Отсюда можно сделать два важных вывода:

1. Ближняя плоскость отсечения должна быть как можно дальше от наблюдателя (как можно ближе к объектам сцены, насколько это позволяет логика работы графического приложения), чтобы область глубин высокой точности не пустовала, а объекты первого плана не попадали в зону пониженной точности.
2. Протяжённость сцены, т. е. $z_{far} - z_{near}$ должна быть как можно меньше (опять же, насколько это позволяет логика работы).

Об этом всегда нужно помнить, проектируя трёхмерные сцены, чтобы минимизировать вероятность возникновения ситуации борьбы глубин. Для объектов, удалённых от наблюдателя не дальше, чем на z, минимальное «безопасное» значение расстояние Δ, на

котором они должны находится друг от друга, чтобы не вступить в борьбу, можно грубо оценить по формуле

$$\Delta = \frac{z^2}{2^n \, z_{near} - z}.$$

Здесь предполагается, что $\Delta \ll z$ и $z_{near} \ll z_{far}$.

Буфер глубины может быть получен из видеопамяти в основное приложение для осуществления каких-либо операций, требующих знания фактических значений глубины сцены в каждой точке изображения. Например, на основе буфера глубины может быть организован эффект переменной резкости, когда определённая точка на сцене полагается точкой фокусировки вирутальной камеры, и части изображения, глубина которых отличается от глубины точки фокуса, размываются пропорционально модулю разности этих глубин.

Содержимое буфера глубины может быть визуализировано в виде изображения в градациях серого (значения интерпретируется как яркость). Результат такой визуализации показан на рис. 19.

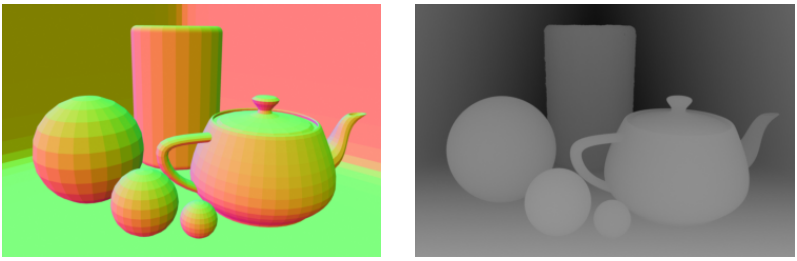

Рис. 19. Трёхмерная сцена (слева) и соответствующее ей содержание буфера глубины (справа)

В OpenGL тест глубины может быть в любой момент включен командой `glEnable(GL_DEPTH_TEST)` и отключен командой `glDisable(GL_DEPTH_TEST)`. Для работы теста глубины, однако, необходимо, чтобы заранее был создан буфер глубины и связан с буфером кадра.

Отсутствие теста глубины при визуализации трёхмерной сцены приводит к достаточно «сюрреалистичным» результатам:

поскольку перекрытия в этом случае зависят исключительно от порядка вывода полигонов, некоторые более близкие к наблюдателю объекты оказываются перекрыты более дальними. Кроме того, часть объектов может казаться «вывернутыми наизнанку», поскольку их «задние» грани вывелись позже «передних».

10.2. Визуализация трёхмерного объекта

Одним из самых тривиальных трёхмерных объектов является куб, поэтому его и предлагается избрать для тестирования. Для простоты можно взять куб $[-1, 1] \times [-1, 1] \times [-1, 1]$. Каждую сторону предлагается раскрасить в свой цвет, чтобы они хорошо отличались друг от друга. Геометрически куб содержит 8 вершин, однако в этом случае из-за интерполяции цвета у него не будет выраженных углов (цвет разных граней будет смешиваться в углах). Поэтому придётся продублировать часть вершин, исключив тем самым смешивание. Каждая сторона будет представлена 4 вершинами, составляющими 2 треугольника, итого 24 вершины и 12 треугольников.

Индексация вершин представлена на рис. 20. Эта индексация удовлетворяет описанному ранее условию, что у полигонов, повёрнутых внешней стороной к наблюдателю, обход вершин осуществляется против часовой стрелки.

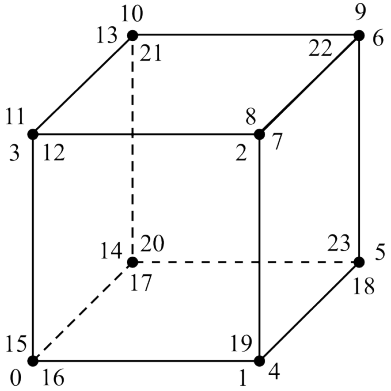

Рис. 20. Индексация вершин куба

Формат атрибутов вершин предлагается такой: (x, y, z, r, g, b). Соответствующие вершинный и индексный массивы представлены в листинге 9.

Листинг 9. Массивы вершин и индексов куба

```
const GLfloat vertices[] =
{
    −1.0, −1.0, 1.0, 1.0, 0.0, 0.0,
    1.0, −1.0, 1.0, 1.0, 0.0, 0.0,
    1.0, 1.0, 1.0, 1.0, 0.0, 0.0,
    −1.0, 1.0, 1.0, 1.0, 0.0, 0.0,

    1.0, −1.0, 1.0, 1.0, 1.0, 0.0,
    1.0, −1.0, −1.0, 1.0, 1.0, 0.0,
    1.0, 1.0, −1.0, 1.0, 1.0, 0.0,
    1.0, 1.0, 1.0, 1.0, 1.0, 0.0,

    1.0, 1.0, 1.0, 1.0, 0.0, 1.0,
    1.0, 1.0, −1.0, 1.0, 0.0, 1.0,
    −1.0, 1.0, −1.0, 1.0, 0.0, 1.0,
    −1.0, 1.0, 1.0, 1.0, 0.0, 1.0,

    −1.0, 1.0, 1.0, 0.0, 1.0, 1.0,
    −1.0, 1.0, −1.0, 0.0, 1.0, 1.0,
    −1.0, −1.0, −1.0, 0.0, 1.0, 1.0,
    −1.0, −1.0, 1.0, 0.0, 1.0, 1.0,

    −1.0, −1.0, 1.0, 0.0, 1.0, 0.0,
    −1.0, −1.0, −1.0, 0.0, 1.0, 0.0,
    1.0, −1.0, −1.0, 0.0, 1.0, 0.0,
    1.0, −1.0, 1.0, 0.0, 1.0, 0.0,

    −1.0, −1.0, −1.0, 0.0, 0.0, 1.0,
    −1.0, 1.0, −1.0, 0.0, 0.0, 1.0,
    1.0, 1.0, −1.0, 0.0, 0.0, 1.0,
    1.0, −1.0, −1.0, 0.0, 0.0, 1.0
```

```
};

const GLuint indices[] =
{
    0, 1, 2, 2, 3, 0,
    4, 5, 6, 6, 7, 4,
    8, 9, 10, 10, 11, 8,
    12, 13, 14, 14, 15, 12,
    16, 17, 18, 18, 19, 16,
    20, 21, 22, 22, 23, 20
};
```

В данном случае уже не обойтись без преобразований вершин: во-первых, созданная модель занимает собой весь куб NDC, во-вторых, для реалистичного отображения трёхмерной сцены нужна перспективная проекция, в-третьих, для более наглядного отображения куба ему нужно задать некоторый угол поворота, иначе он спроецируется в квадрат.

В связи с этим необходимо изменить вершинный шейдер, добавив в него юниформ-переменную, хранящую матрицу преобразования, и осуществив само преобразование. Код шейдеров с учётом указанных изменений представлен в листинге 10.

Листинг 10. Код шейдеров для вывода куба

```
const GLchar vsh[] =
"#version 330\n"
""
"layout(location = 0) in vec3 a_position;"
"layout(location = 1) in vec3 a_color;"
""
"uniform mat4 u_pm;"
""
"out vec3 v_color;"
""
"void main()"
"{"
```

```
13 "    v_color = a_color;"
14 "    gl_Position = u_pm * vec4(a_position, 1.0);"
15 "}";
16
17 const GLchar fsh[] =
18 "#version 330\n"
19 ""
20 "in vec3 v_color;"
21 ""
22 "layout(location = 0) out vec4 o_color;"
23 ""
24 "void main()"
25 "{"
26 "   o_color = vec4(v_color, 1.0);"
27 "}";
```

Вершинный шейдер ожидает получить из основной программы матрицу PM, то есть произведение матрицы проекции и матрицы модели (результат перемножения всех матриц размещения). Умножение этих матриц часто осуществляется в основной программе, а не в шейдере, поскольку в основной программе оно произойдёт однократно для всего массива вершин, тогда как в шейдере производилось бы заново для каждой обрабатываемой вершины.

Поскольку в нашем случае используется GLSL версии 330, нет возможности явным образом задать дескриптор юниформа и придётся запрашивать его программно. Для этого необходимо создать глобальную переменную типа `GLint` и присвоить ей результат функции `glGetUniformLocation(g_shaderProgram, "u_pm")`, где `g_shaderProgram` – дескриптор шейдерной программы.

Для включения теста глубины в функцию `init` (отвечающую за инициализацию основных структур данных сцены и начальную настройку состояния конвейера) необходимо добавить вызов `glEnable(GL_DEPTH_TEST)`. Для заполнения буфера глубины начальными значениями необходимо изменить вызов очистки на `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);`

Функция `glClear` принимает в качестве аргумента битовую маску буферов, которые следует очистить, что позволяет минимизировать число вызовов.

Затем необходимо подготовить матрицу преобразования. В данном случае она может храниться глобально, но часто её окончательный вид определяется непосредственно перед отрисовкой каждого нового кадра (в особенности, если на сцене имеет место анимация трансформаций).

Конкретный код формирования матрицы зависит от используемой матричной библиотеки. В качестве матрицы проекции предлагается использовать перспективное преобразование с углом обзора $\pi/4$, ближней плоскостью отсечения на расстоянии 0.1 и дальней – на расстоянии 10. В качестве матрицы модели предлагается использовать произведение матрицы переноса на -5 вдоль оси OZ, поворота на $-\pi/4$ вокруг оси OX и поворота на $\pi/4$ вокруг оси OY.

Цель этих преобразований – «отодвинуть» куб от наблюдателя и обеспечить ракурс, в котором было бы видно, что это объёмный, а не плоский объект.

Подготовленная матрица должна быть отправлена в шейдер. Для этого может быть использован вызов вида

```
glUniformMatrix4fv(g_uMP, 1, GL_FALSE, g_mp.data());
```

Он должен быть помещён непосредственно перед вызовом отрисовки. Здесь `g_uMP` – дескриптор соответствующего юниформа, `g_mp.data()` – метод, возвращающий линейный массив данных матрицы итогового преобразования.

Следует отметить, что для означивания юниформов в OpenGL существуют функции «на все случаи жизни»: принимающие целые числа, числа с плавающей точкой, векторы и матрицы разной размерности. С этими функциями читателю предлагается ознакомиться самостоятельно в документации OpenGL [11].

Задание: Изменить тестовое приложение так, чтобы на экран был выведен куб в перспективной проекции. Изменить проекцию на параллельную. Сравнить результаты.

10.3. Анимация преобразований

Для того чтобы корректно организовать анимационное изменение каких-либо свойств объекта, как отмечалось ранее, необходимо знать время, затрачиваемое на визуализацию одного кадра. Самый простой способ сделать это – воспользоваться системной функцией запроса текущего времени и каждый кадр производить измерение. Для обеспечения независимости от платформы удобно использовать абстракцию, предоставляемую `std::chrono`. Пример мультиплатформенной реализации такого подхода представлен в листинге 11.

Листинг 11. Измерение времени для организации анимации

```
g_callTime = chrono::system_clock::now();

while (glfwGetKey(g_window, GLFW_KEY_ESCAPE) !=
       GLFW_PRESS &&
       glfwWindowShouldClose(g_window) == 0)
{
    auto callTime = chrono::system_clock::now();
    chrono::duration<double> elapsed = callTime -
                                       g_callTime;
    g_callTime = callTime;
    draw(elapsed.count());

    glfwSwapBuffers(g_window);
    glfwPollEvents();
}
```

Глобальная переменная, хранящая время предыдущего вызова, описана как

```
chrono::time_point<chrono::system_clock> g_callTime;
```

Для использования типов и функций, связанных с изменением времени, необходимо подключить заголовочный файл `chrono`.

Функция `draw`, которая раннее не имела параметров, теперь принимает параметр `double delta` – время, *затраченное на визу-*

ализацию предыдущего кадра. Ограничением такого подхода является неверное определение времени самого первого кадра (в данном случае `delta` в первом кадре будет близка к нулю, тогда как в последующих кадрах, при условии включенной вертикальной синхронизации, она, скорее всего, будет около $1/60$). Следовательно анимация, как и графический конвейер, тоже нуждается в «разогреве».

Задание: Добавить в приложение анимацию изменения угла поворота куба. Добавить в приложение анимацию изменения цвета куба.

Подсказка: Для изменения цвета необходимо внести в один из шейдеров некоторую юниформ-переменную, являющуюся параметром для вычисления цвета, и изменять значение этой переменной в основной программе в соответствии с параметром `delta`.

11. Отладка графического приложения

Пожалуй, единственным действительно слабым местом OpenGL являются возможности, предоставляемые для отладки. Основной механизм здесь состоит в запросе специального поля ошибки машины состояний, который осуществляется функцией `glGetError`. Эта функция возвращает целочисленный код возникшей ошибки либо 0, если ошибок не было, и обнуляет поле ошибки.

Различные API-функции OpenGL могут записывать в поле ошибки значения, в той или иной степени характеризующие природу ошибочной ситуации. В документации каждой API-функции расписан набор возможных кодов ошибок, связанных с ней, и *примерно* объяснены условия возникновения соответствующих проблем. Однако число самих кодов весьма ограничено, они активно переиспользуются, из-за чего пропадает однозначность раскодирования. Сказать наверняка, к какой функции относится то или иное значение поля ошибки, можно лишь в том случае, если запрос кода происходит непосредственно после каждого вызова. Более того, даже зная, какая функция записала данный код, далеко не всегда можно понять, что же именно привело к этой ошибке и как её исправить.

При этом достаточно часто на практике возникают ситуации, когда «формально» ошибок нет, однако результат визуализации некорректен. Как правило, это связано с неверным размещением объектов, неверными вершинными и индексными массивами, ошибками в алгоритмах шейдеров и т. д. Самым неприятным всегда бывает увидеть пустой экран вместо желаемой сцены, поскольку сходу обычно совершенно не понятно, какое звено в цепочке преобразований данных на графическом конвейере сработало неверно. Существуют профилировщики графического конвейера OpenGL, однако почти полностью отсутствуют какие-либо инструменты

отладки, которые бы, например, позволяли осуществлять удобную трассировку конвейера.

Отладка кода шейдеров в OpenGL стандартными средствами невозможна: нельзя указывать точки останова, осуществлять трассировку или хотя бы выводить куда-либо значения переменных (нет аналогов `std::cout`). Единственной возможностью хоть как-то увидеть значение переменной является конвертация его в цвет обрабатываемой поверхности (нормировка и назначение его в качестве одного из компонентов цвета главной цели рендеринга).

При работе с программируемым конвейером следует помнить об очень широком спектре существующих графических процессоров. Различные специальные функции могут быть доступны на одних процессорах, но недоступны на других, поэтому при реализации сложных алгоритмов следует при помощи соответствующих методов проверять, поддерживаются ли запрашиваемые механизмы на текущей платформе (в особенности это касается т.н. расширений – функций, не входящих в описанное спецификацией ядро графического API).

Более того, из-за аппаратных особенностей на различном оборудовании некоторые функции могут иногда работать по-разному. Источник ошибок такого рода очень трудно обнаружить, поэтому рекомендуется внимательно читать документацию на используемые функции, даже если принцип их работы кажется на первый взгляд очевидным. Так, например, по спецификации GLSL результат функции `pow(x,y)`, вычисляющей $f(x, y) = x^y$, не определён, если $x < 0$. При этом на большинстве графических процессоров выражение вида `a = pow(-2.0, 2.0)` будет вычислено правильно, но есть оборудование, на котором результатом станет `NaN`.

Точки останова, как правило, доступны в основном приложении, однако тоже далеко не всегда помогают в отладке: например, в контексте поиска ошибок в анимации, когда происходит визуализация длинной последовательности кадров, и критичным является реальный масштаб времени. В этом случае останов программы резко увеличивает время, затрачиваемое на кадр, и тем самым сбивает ход анимации. Кроме того, многократные остановы внутри цикла визуа-

лизации, как правило, неинформативны. В таких ситуациях адекватным решением является вывод значений интересующих переменных на консоль (или в файл) и последующий анализ изменения этих значений с течением времени.

Наиболее общей рекомендацией при отладке графического приложения является метод «разделяй и властвуй». В случае обнаружения ошибки неизвестной природы следует постепенно упрощать сцену вплоть до нахождения действий, непосредственно приводящих к ошибке. После каждого, пусть даже незначительного изменения работающего кода, следует производить тестирование корректности визуализации.

Однако есть надежда, что удобные средства для отладки преобразований на конвейере OpenGL всё-таки появятся в ближайшем будущем, поскольку, например, для Direct3D они уже существуют [12].

Заключение

Достаточно подробное введение в компьютерную графику можно считать законченным. Следует, однако, ещё раз подчеркнуть, что это было именно введение, а разобранное по ходу повествования приложение – своего рода «Hello World».

У современных низкоуровневых графических API достаточно высокий «порог вхождения» для новичков. Посудите сами: для того чтобы вывести один цветной треугольник на экран, пришлось как минимум написать три разных программы (основную, вершенный и фрагментный шейдеры) для двух разных процессоров (CPU и GPU), а как максимум – ещё и разобраться в тонкостях устройства графического конвейера и математического аппарата вычислительной геометрии. Однако этот порог можно считать пройденным.

В качестве дальнейших шагов в изучении компьютерной графики предлагается ознакомится, например, с такими компетентными источниками, как «OpenGL Суперкнига» [1] и «Курс уроков Neon-Helium» [2].

Список литературы

1. Sellers G., Right Jr. R. S., Haemel N. OpenGL Superbible. Seventh Edition. Pearson Education, Inc. 2015.
2. Neon-Helium [Электронный ресурс]. URL: http://nehe.gamedev.net/ (дата обращения: 12.04.2017).
3. Библиотека Eigen [Электронный ресурс]. URL: http://eigen.tuxfamily.org/ (дата обращения: 12.04.2017).
4. Библиотека OpenSceneGraph [Электронный ресурс]. URL: http://www.openscenegraph.org/ (дата обращения: 12.04.2017).
5. Библиотека GLM [Электронный ресурс]. URL: http://glm.g-truc.net/ (дата обращения: 12.04.2017).
6. Keval H., Sasse M. A. To catch a thief – you need at least 8 frames per second: the impact of frame rates on user performance in a CCTV detection task // Proceedings of the 16th ACM international conference on Multimedia. ACM, 2008. P. 941–944.
7. Библиотека GLEW [Электронный ресурс]. URL: http://glew.sourceforge.net/ (дата обращения: 12.04.2017).
8. Библиотека GLFW [Электронный ресурс]. URL: http://www.glfw.org/ (дата обращения: 12.04.2017).
9. The United States Patent and Trademark Office. Retina [Электронный ресурс]. 2011. URL: http://tsdr.uspto.gov/#caseNumber=85056807&caseType=SERIAL_NO&searchType=statusSearch (дата обращения: 12.04.2017).
10. Документация языка GLSL [Электронный ресурс]. URL: https://khronos.org/registry/OpenGL/index_gl.php (дата обращения: 12.04.2017).
11. Документация стандарта OpenGL [Электронный ресурс]. URL: https://www.opengl.org/sdk/docs/man/ (дата обращения: 12.04.2017).
12. Средства отладки Direct3D [Электронный ресурс]. URL: https://msdn.microsoft.com/en-us/library/hh873193.aspx (дата обращения: 12.04.2017).

Учебное издание

Рябинин Константин Валентинович

ВЫЧИСЛИТЕЛЬНАЯ ГЕОМЕТРИЯ И АЛГОРИТМЫ КОМПЬЮТЕРНОЙ ГРАФИКИ

РАБОТА С 3D-ГРАФИКОЙ СРЕДСТВАМИ OPENGL

Учебное пособие

Редактор *М. А. Шемякина*
Корректор *Н. А. Антонова*
Компьютерная вёрстка: *К. В. Рябинин*

Подписано в печать 12.04.2017. Формат 60x84/16.
Усл. печ. л. 5,81. Тираж 100 экз. Заказ__58__

Издательский центр
Пермского государственного
национального исследовательского университета.
614990, г. Пермь, ул. Букирева, 15

Типография ПГНИУ.
614990, г. Пермь, ул. Букирева, 15