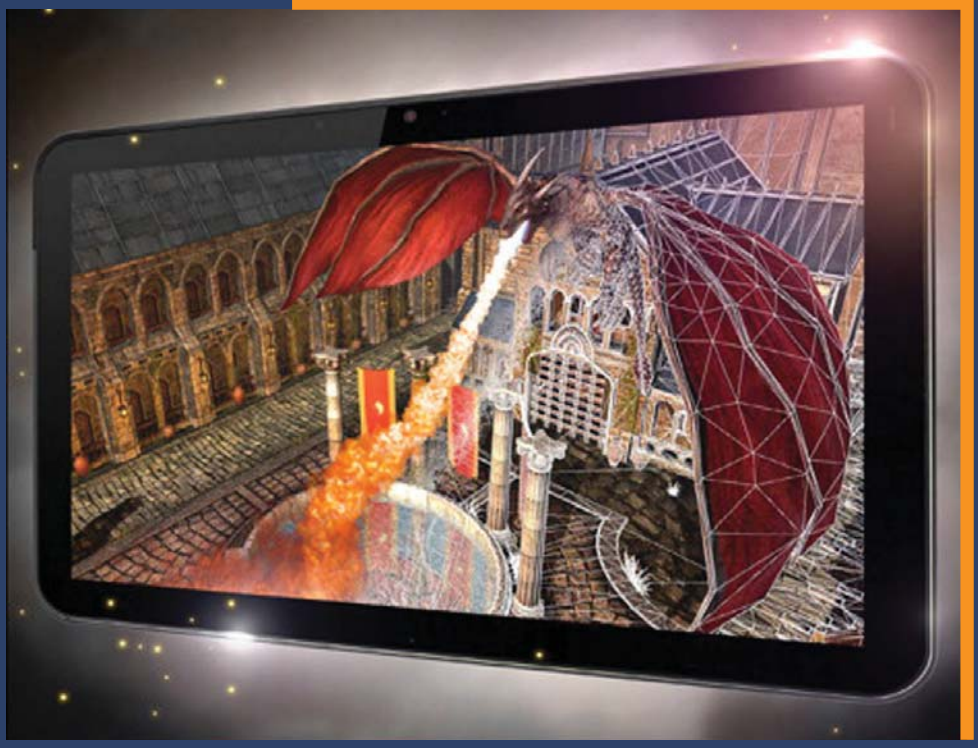
ДМК ПРЕСС ИЗДАТЕЛЬСТВО

OpenGL ES 3.0

Руководство разработчика



Дэн Гинсбург ■ Будирижанто Пурномо

при участии Дейва Шрейнера и Аафтаба Мунши

УДК 004'27OpenGL ES 3
ББК 32.972.121
Г49

Гинсбург Д., Пурномо Б.
Г49 OpenGL ES 3.0. Руководство разработчика / пер. с англ. А. Борескова. – М.: ДМК Пресс, 2015. – 448 с.: ил.

ISBN 978-5-97060-256-0

OpenGL ES – это ведущий интерфейс и графическая библиотека для рендеринга сложной трехмерной графики на мобильных устройствах. Последняя версия, OpenGL ES 3.0, делает возможным создания потрясающей графики для новых игр и приложений, не влияя на производительность устройства и время работы аккумулятора.

В данной книге авторы рассматривает весь API и язык для написания шейдеров. Они внимательно рассматривают возможности OpenGL ES такие как теневые карты, дублирование геометрии, рендеринг в несколько текстур, uniform-буферы, сжатие текстур, бинарное представление программ и преобразование обратной связи. Шаг за шагом вы перейдете от вводных примеров к продвинутому попиксельному освещению и системам частиц. Также вы найдете содержательные советы по оптимизации быстродействия, максимизации эффективности работы API и GPU и полном использовании OpenGL ES в широком спектре приложений.

На сайте издательства www.dmkpress.com выложены примеры к книге на языке C.

Издание предназначено программистам мобильных приложений, желающих максимально использовать графические возможности своих устройств.

УДК 004'27OpenGL ES 3
ББК 32.972.121

ISBN 978-0-321-93388-1 (анг.)
ISBN 978-5-97060-256-0 (рус.

Содержание

Глава 1

Введение в OpenGL ES 3.0

OpenGL для встроенных систем (OpenGL ES) – это API для продвинутой 3D-графики, рассчитанный на мобильные и встроенные устройства. OpenGL ES – это основной графический API для современных смартфонов, применяется даже на настольных системах. Список платформ, поддерживающих OpenGL ES, включает в себя iOS, Android, BlackBerry, bada, Linux и Windows. OpenGL ES также лежит в основе WebGL – стандарта для трехмерной графики для веба.

Начиная с релиза iPhone 3GS в июне 2009-го и Android 2.0 в марте 2010-го, OpenGL ES 2.0 поддерживался на устройствах с iOS и Android. Первое издание этой книги детально рассматривало OpenGL ES 2.0. Нынешнее издание нацелено на OpenGL ES 3.0, следующую версию OpenGL ES. Практически неизбежно, что каждая развивающаяся мобильная платформа будет поддерживать OpenGL ES 3.0. Сейчас OpenGL ES 3.0 уже поддерживается на устройствах с Android 4.3 и старше и на iPhone 5S с iOS7. OpenGL ES 3.0 обратно совместим с OpenGL ES 2.0, под этим подразумевается, что приложения, написанные под OpenGL ES 2.0, будут работать и под OpenGL ES 3.0.

OpenGL ES – это один из многих API, созданных Khronos Group. Основанная в январе 2000 года, Khronos Group – это поддерживаемый своими членами консорциум, ориентированный на создание открытых стандартов и API, не требующих лицензирования. Khronos Group также занимается развитием OpenGL – кроссплатформенного API для трехмерной графики, ориентированного на настольные системы с Linux, различными вариантами UNIX, Mac OS X и Microsoft Windows. Это широко распространенный 3D API, имеющий большое влияние в мире.

В связи с широким распространением OpenGL как API для трехмерной графики его положили в основу при разработке открытого стандарта для 3D-графики для мобильных и встроенных устройств и затем изменили для того, чтобы соответствовать потребностям и ограничениям мобильных и встроенных устройств. В ранних версиях OpenGL ES (1.0, 1.1, 2.0) эти ограничения включали в себя ограниченные возможности по обработке данных, невысокую ширину шины для передачи данных и чувствительность к потреблению энергии. При определении спецификаций OpenGL ES рабочая группа использовала следующие критерии:

- OpenGL API очень большой и сложный, а целью рабочей группы по OpenGL ES является создание API, подходящего для устройств с ограниченными возможностями. Для достижения этой цели рабочая группа убрала всю избыточность из OpenGL API. В случае, когда одна и та же операция может быть выполнена более, чем одним способом, оставлялся наиболее полезный способ, остальные просто удалялись. Хорошим примером этого является за-

дание геометрии, когда в OpenGL приложение может использовать непосредственный режим, дисплейные списки и вершинные массивы. В OpenGL ES существуют только вершинные массивы; непосредственный режим и дисплейные списки были удалены.

- Удаление избыточности является важной целью, но также важным было сохранение совместимости с OpenGL. По возможности, OpenGL ES был разработан так, что приложения, написанные на подмножестве OpenGL для встроенных систем, также будут работать на OpenGL ES. Это было важной целью, поскольку позволило бы разработчикам использовать сразу оба API и разрабатывать приложения и инструменты, применяющие общую функциональность.
- Новые возможности были добавлены для учета специфических ограничений мобильных и встроенных устройств. Например, для уменьшения потребления энергии и увеличения быстродействия шейдеров в язык для написания шейдеров были добавлены спецификаторы точности.
- Разработчики OpenGL ES хотели гарантировать минимальный набор возможностей для обеспечения качества получаемых изображений. В ранних мобильных устройствах размеры экрана были довольно небольшими, поэтому было важно, чтобы качество каждого выводимого пиксела было настолько хорошим, насколько это возможно.
- Рабочая группа по OpenGL ES хотела гарантировать, что любая реализация OpenGL ES будет удовлетворять определенным соглашениям по качеству изображения, корректности и устойчивости. Для этого были разработаны соответствующие тесты, которые должны быть выполнены для того, чтобы реализация OpenGL ES была признана совместимой.

На данный момент времени Khronos выпустил четыре спецификации OpenGL ES: OpenGL ES 1.0 и OpenGL ES 1.1 (в книге мы будем обозначать их как OpenGL ES 1.x), OpenGL ES 2.0 и OpenGL ES 3.0. Спецификации OpenGL ES 1.0 и 1.1 реализуют фиксированный конвейер рендеринга и основаны, соответственно, на спецификациях OpenGL 1.3 и OpenGL 1.5.

Спецификации OpenGL ES 2.0 вводят программируемый конвейер и основаны на спецификациях OpenGL 2.0. Это значит, что соответствующие спецификации для OpenGL были взяты за основу для определения того, что войдет в соответствующую версию OpenGL ES.

OpenGL ES 3.0 – это следующий шаг в эволюции мобильной графики, он основан на спецификациях OpenGL 3.3. В то время когда OpenGL ES 2.0 был успешен в привнесении на мобильные устройства возможностей, похожих на DirectX 9 и Microsoft Xbox 360, на настольных системах графика продолжала развиваться. В OpenGL ES 2.0 не хватало многих важных возможностей, необходимых для реализации таких эффектов, как теневые карты, рендеринг объемных фигур (volume rendering), выполняемая на GPU анимация систем частиц, instancing, сжатие текстур и гамма-коррекция. OpenGL ES 3.0 добавляет эти возможности для мобильных устройств, продолжая при этом философию адаптации к ограничениям мобильных устройств.

Конечно, некоторые из этих ограничений, учтенные при разработке предыдущих версий OpenGL ES, были уже не актуальны. Например, на мобильных устройствах сейчас экраны с большим размером (иногда их разрешение даже выше, чем на настольных системах). Кроме того, на многих мобильных устройствах сейчас есть многоядерные центральные процессоры и большой объем памяти. Поэтому при разработке OpenGL ES 3.0 целью Khronos стал, скорее, своевременный вывод на рынок новых возможностей, а не ограниченные возможности устройств.

Следующие разделы посвящены конвейеру OpenGL ES.

OpenGL ES 3.0

Как уже отмечалось, эта книга посвящена OpenGL ES 3.0 API. Нашей целью является детально разобрать спецификации OpenGL ES 3.0, дать примеры использования OpenGL ES 3.0 и обсудить различные способы оптимизации. После прочтения этой книги у вас будет отличное понимание OpenGL ES 3.0 API, вы сможете легко писать сложные приложения, использующие OpenGL ES 3.0, и вам не придется изучать многочисленные спецификации, для того чтобы понять, как та или иная возможность работает.

OpenGL ES 3.0 реализует программируемый графический конвейер и состоит из двух частей – описания OpenGL ES 3.0 API и описания языка шейдеров для OpenGL ES 3.0. На рис. 1.1 приведен графический конвейер OpenGL ES 3.0. Закрашенные прямоугольники на этом рисунке обозначают программируемые части конвейера в OpenGL ES 3.0. Далее приводится обзор каждой из частей конвейера.

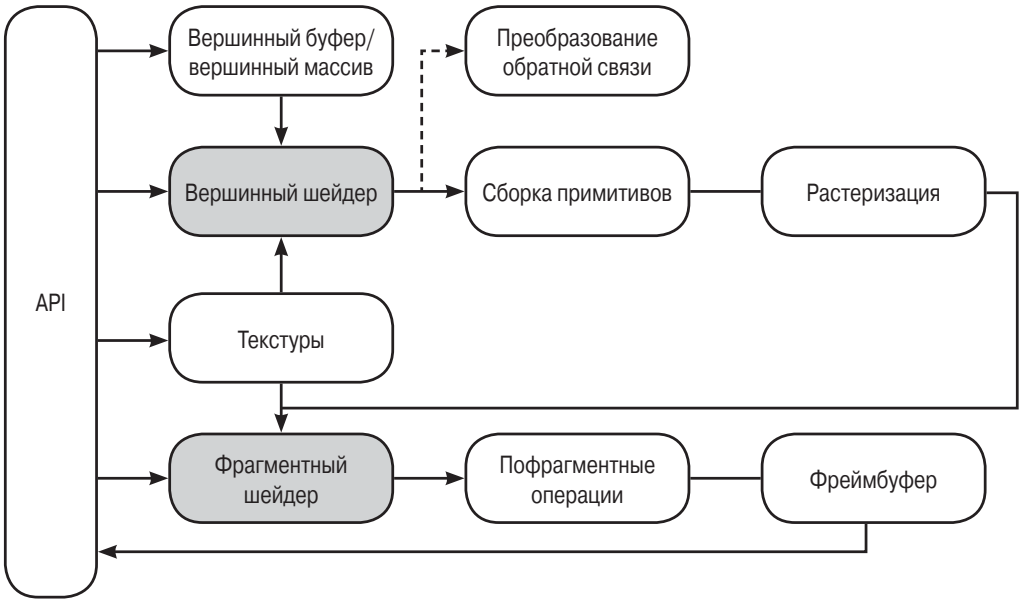


Рис. 1.1 ❖ Графический конвейер OpenGL ES 3.0

Вершинный шейдер

Этот раздел дает обзор вершинных шейдеров. Детально вершинные и фрагментные шейдеры будут рассмотрены в дальнейших главах книги. Вершинный шейдер реализует программируемый подход для работы с вершинами.

Входными данными для вершинного шейдера являются:

- исходный код шейдера в виде теста или в бинарном виде, задающий действия, которые будут выполняться над вершиной;
- входные значения вершины (атрибуты) – данные, передаваемые для каждой вершины при помощи вершинных массивов;
- Uniform-переменные – значения, используемые вершинными и фрагментными шейдерами;
- сэмплеры – особый тип uniform-переменных, служащий для представления текстур, используемых вершинным шейдером.

Выходные значения вершинного шейдера назывались в OpenGL ES 2.0 varying-переменными, но в OpenGL ES 3.0 были переименованы в выходные переменные. На стадии растеризации примитивов выходные значения вершинного шейдера вычисляются для каждого полученного фрагмента и передаются как входные значения во фрагментный шейдер. Механизм, используемый для получения значений для каждого фрагмента из выходных значений вершинного шейдера для вершин, называется интерполяцией. Также OpenGL ES 3.0 добавляет новую возможность, называемую преобразованием обратной связи (transform feedback), которая позволяет записать выходные значения вершинного шейдера в буфер (в дополнение или вместо обработки фрагментным шейдером). Например, как показано в примере по преобразованию обратной связи в главе 14, система частиц может быть реализована при помощи вершинного шейдера, в котором частицы выводятся в буферный объект при помощи преобразования обратной связи. Входные и выходные значения вершинного шейдера показаны на рис. 1.2.

Вершинные шейдеры могут быть использованы для традиционных операций над вершинами, так как преобразование координат при помощи матриц, вычисление при помощи уравнения освещенности цвета в вершине, генерирование или преобразование текстурных координат. Кроме этого, поскольку вершинный шейдер задается в приложении, вершинные шейдеры могут быть использованы для реализации нестандартных преобразований, освещения или повершинных эффектов, недоступных в традиционных фиксированных конвейерах.

Пример 1.1 показывает вершинный шейдер, написанный с использованием языка шейдеров OpenGL ES. Мы детально объясним вершинные шейдеры позже. Мы приводим этот шейдер здесь для того, чтобы дать вам представление о том, как выглядит вершинный шейдер. Вершинный шейдер из примера 1.1 берет положение и связанный с ним цвет как входные атрибуты, преобразует координаты при помощи матрицы 4×4 и выводит преобразованные координаты и цвет.

Строка 1 задает версию шейдерного языка – информацию, которая должна быть в первой строке шейдера (`#version 300 es` задает использование шейдерного языка для OpenGL ES 3.0). Строка 2 описывает uniform-переменную `u_mvpMatrix`,

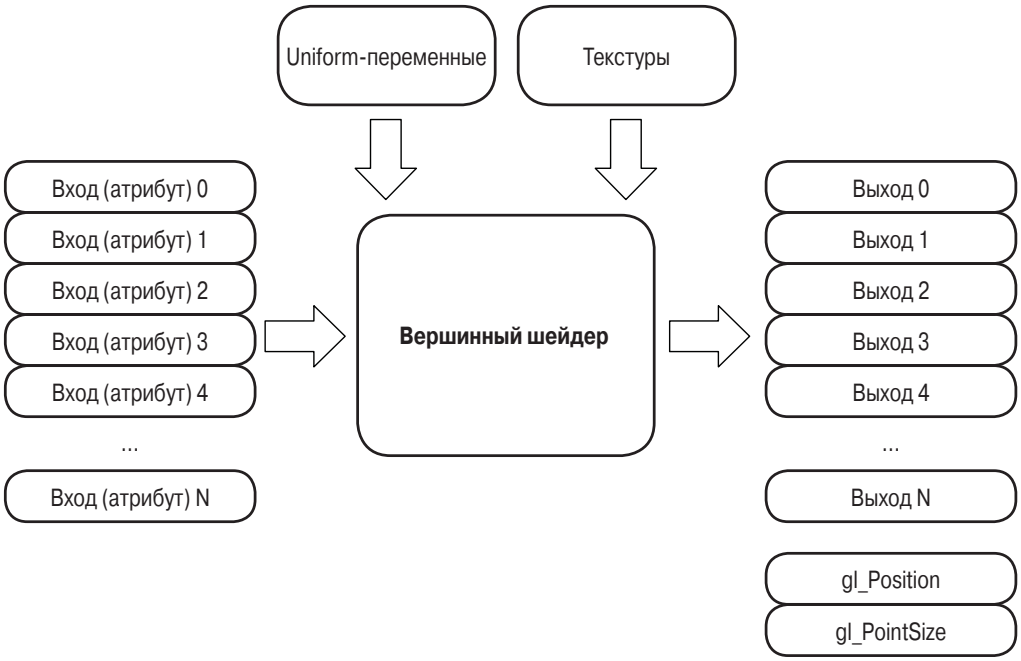


Рис. 1.2 ❖ Вершинный шейдер OpenGL ES

Пример 1.1 ❖ Пример вершинного шейдера

```
1.  #version 300 es
2.  uniform mat4 u_mvpMatrix; // matrix to convert a_position
3.                            // from model space to normalized
4.                            // device space
5.
6.  // attributes input to the vertex shader
7.  in vec4 a_position;       // position value
8.  in vec4 a_color;          // input vertex color
9.
10. // output of the vertex shader - input to fragment
11. // shader
12. out vec4 v_color;         // output vertex color
13. void main()
14. {
15.    v_color = a_color;
16.    gl_Position = u_mvpMatrix * a_position;
17. }
```

которая хранит в себе произведение модельно-видовой матрицы и матрицы проектирования. Строки 7 и 8 описывают входные значения для вершинного шейдера, называемые вершинными атрибутами. `a_position` – это атрибут, соответствую-

щий координатам вершины, а a_color – это атрибут, содержащий цвет вершины. В строке 12 мы описываем выходное значение v_color, в которое мы запишем цвет в вершине. Встроенная переменная gl_Position в объявлении не нуждается, и вершинный шейдер должен записать преобразованные координаты в эту переменную. У вершинного и фрагментного шейдеров есть всего одна точка входа – функция main. Строки 13–17 задают функцию main. В строке 15 мы читаем атрибут вершины из a_color и записываем его в выходную переменную v_color. В строке 16 преобразованные координаты вершины записываются в переменную gl_Position.

Сборка примитивов

После вершинного шейдера следующей стадией конвейера OpenGL ES 3.0 является сборка примитивов. Примитив – это геометрический объект, такой как треугольник, отрезок или точечный спрайт. Каждая вершина примитива посылается отдельной копии вершинного шейдера. Во время сборки примитива эти вершины группируются вместе для образования примитива.

Для каждого примитива необходимо определить, лежит ли он внутри области видимости (области трехмерного пространства, которая видна на экране). Если примитив не полностью находится внутри области видимости, то необходимо выполнить отсечение примитива по границе области видимости. Если примитив находится полностью вне области видимости, то он отбрасывается. После отсечения координаты вершин переводятся в координаты на экране. Также может быть произведено отбрасывание граней в зависимости от того, какой стороной они повернуты. После отсечения и отбрасывания примитив готов для передачи на следующую стадию конвейера – стадию растеризации.

Растеризация

Следующая стадия, показанная на рис. 1.3, – это стадия растеризации, на которой соответствующий примитив (точечный спрайт, отрезок или треугольник) выводится. Растеризация – это процесс, который переводит примитивы в набор двухмерных фрагментов, обрабатывающихся фрагментным шейдером. Эти двухмерные фрагменты представляют пикселы, которые могут быть нарисованы на экране.

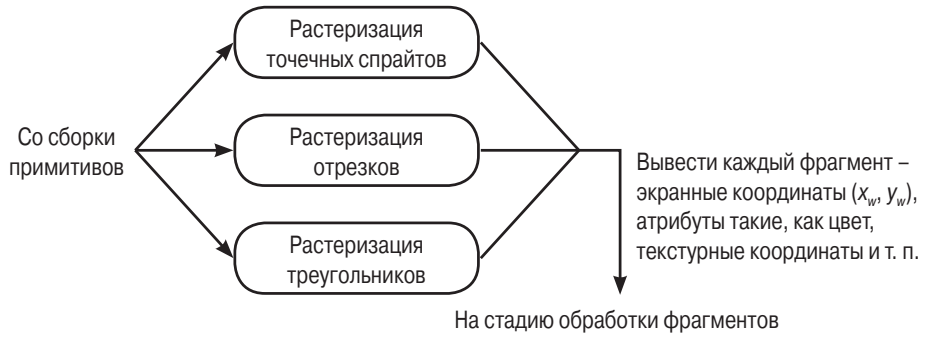


Рис. 1.3 ❖ Стадия растеризации OpenGL ES 3.0

Фрагментный шейдер

Фрагментный шейдер реализует программируемый подход к обработке фрагментов. Как показано на рис. 1.4, этот шейдер выполняется для каждого фрагмента, полученного в ходе стадии растеризации, и получает следующие данные на вход:

- Шейдер в виде исходного текста или бинарного образа, задающего операции, которые необходимо выполнить над фрагментом.
- Входные переменные – выходные значения вершинного шейдера, полученные в процессе растеризации при помощи интерполяции.
- Uniform-переменные – данные, используемые вершинным и фрагментным шейдерами.
- Сэмплеры – специальный тип uniform-переменных, представляющих текстуры, которые использует фрагментный шейдер.

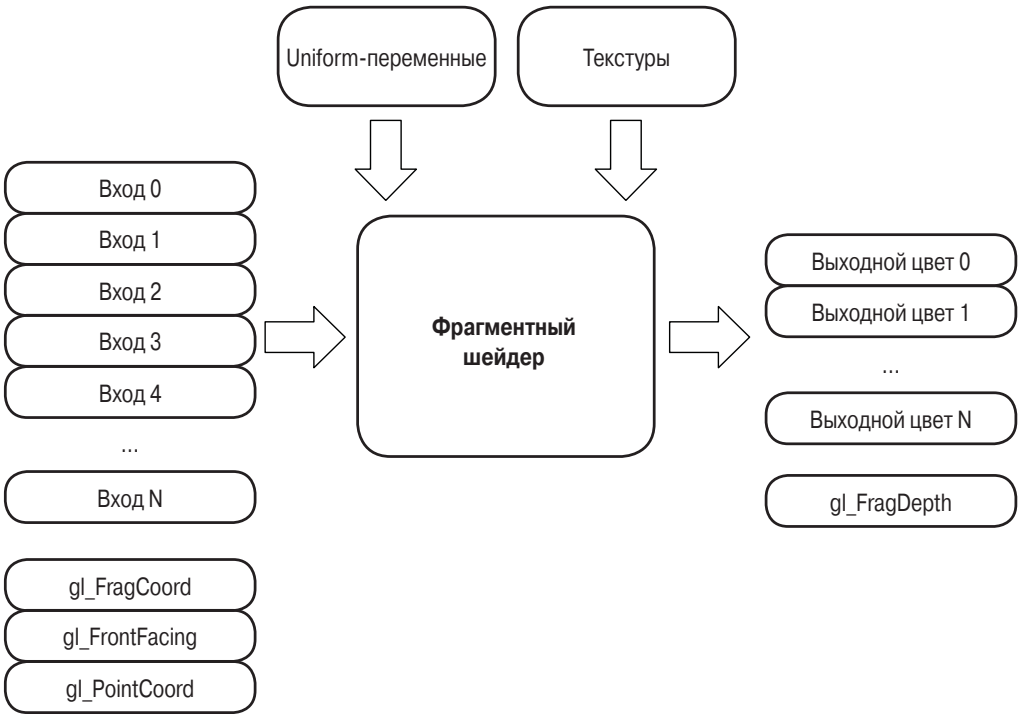


Рис. 1.4 ❖ Фрагментный шейдер OpenGL ES 3.0

Фрагментный шейдер либо отбрасывает фрагмент, либо задает одно или несколько значений цвета, называемых выходными значениями. Обычно выходные значения фрагментного шейдера – это просто один цвет, за исключением рендеринга сразу в несколько текстур (см. соответствующий раздел в главе 11), в последнем случае выводится по одному значению для каждой из текстур, в которую осуществляется рендеринг. Цвет, глубина, значения трафарета и экранные коор-

динаты, полученные на этапе растеризации, становятся входными значениями для стадии пофрагментных операций конвейера рендеринга OpenGL ES 3.0.

Пример 1.2 показывает простой фрагментный шейдер, который может работать вместе с вершинным шейдером из примера 1.1 для вывода треугольника. Детально мы рассмотрим фрагментные шейдеры позже. Здесь этот пример приводится только для того, чтобы дать вам представление о том, как выглядит фрагментный шейдер.

Пример 1.2 ❖ Пример фрагментного шейдера

```
1.  #version 300 es
2.  precision mediump float;
3.
4.  in vec4 v_color;    // input vertex color from vertex shader
5.
6.  out vec4 fragColor; // output fragment color
7.  void main()
8.  {
9.     fragColor = v_color;
10. }
```

Так же, как и в вершинном шейдере, строка 1 задает версию языка для написания шейдеров; эта информация должна быть в первой строке шейдера (`#version 300 es` задает использование языка для написания шейдеров v3.00). Строка 2 задает описатель точности, используемый по умолчанию, это будет разобрано в главе 4. Строка 4 описывает входное значение для фрагментного шейдера. Вершинный шейдер должен записать значения в тот же самый набор переменных, из которых их будет читать фрагментный шейдер. Строка 6 задает описание выходной переменной фрагментного шейдера, которая будет содержать цвет, передаваемый следующей стадии конвейера. Строки 7–10 описывают функцию `main` фрагментного шейдера. Выходное значение берется из переменной `v_color`. Входные значения для фрагментного шейдера линейно интерполируются вдоль примитива перед передачей фрагментному шейдеру.

Пофрагментные операции

После фрагментного шейдера следующей стадией являются пофрагментные операции. Фрагмент, получаемый при растеризации с экранными координатами (x_w, y_w), может изменить во фреймбуфере только пиксел с ординатами (x_w, y_w). Рисунок 1.5 описывает стадии пофрагментных операций в OpenGL ES 3.0.

Во время выполнения стадии пофрагментных операций над фрагментом выполняются следующие функции (и проверки), как показано на рис. 1.5:

- проверка принадлежности пиксела – эта проверка определяет, действительно ли пиксел с координатами (x_w, y_w) принадлежит OpenGL ES. Проверка позволяет оконной системе управлять тем, какие пикселы во фреймбуфере принадлежат текущему контексту OpenGL ES. Например, если окно, показывающее фреймбуфер OpenGL ES, частично закрыто другим окном, то

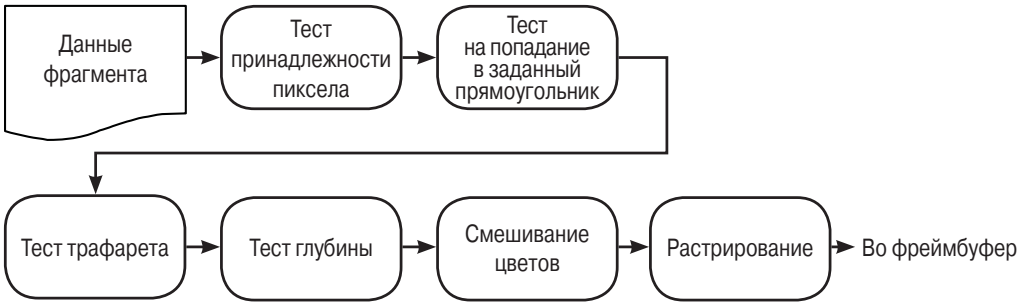


Рис. 1.5 ❖ Пофрагментные операции в OpenGL ES 3.0

оконная система может определить, что закрытые пикселы не принадлежат контексту OpenGL ES и их вообще не надо показывать. В то время как проверка принадлежности является частью OpenGL ES, она не управляется разработчиком, но просто выполняется внутри OpenGL ES;

- проверка ножниц – эта проверка определяет, лежит ли (x_w, y_w) внутри заданного прямоугольника, являющегося частью состояния OpenGL ES. Если фрагмент лежит вне этого прямоугольника, то он отбрасывается;
- проверки трафарета и глубины – эти проверки выполняются над значениями трафарета и глубины фрагмента для определения того, не надо ли отбросить данный фрагмент;
- смешивание – смешивание комбинирует цвет фрагмента с цветом из фреймбуфера по координатам (x_w, y_w);
- растрирование – может быть применено для уменьшения погрешностей, связанных с использованием небольшой точности для хранения цвета во фреймбуфере.

В конце стадии пофрагментных операций либо фрагмент отбрасывается, либо его значения цвета, глубины и трафарета записываются во фреймбуфер по координатам (x_w, y_w). То, какие из этих значений запишутся, зависит от значения соответствующих масок записи. Маски записи дают контроль над записью значений цвета, глубины и трафарета в соответствующие буферы. Например, маска записи для буфера цвета может запретить запись красной компоненты в буфер цвета. Также OpenGL ES 3.0 предоставляет интерфейс для чтения пикселов из фреймбуфера.

Замечание: проверка альфа-компоненты (альфа-тест) и логические операции больше не входят в пофрагментные операции. Они присутствовали в OpenGL ES 2.0 и OpenGL ES 1.x. Поскольку фрагментный шейдер может явно отбрасывать фрагменты, то нет необходимости в выполнении альфа-теста. Логические операции были удалены, так как они крайне редко используются, и рабочая группа не получила пожеланий от производителей по поддержке этой возможности.

Что нового в OpenGL ES 3.0?

OpenGL ES 2.0 открыл эру программируемых шейдеров для мобильных устройств и был крайне успешен при написании игр, приложений и пользовательского интерфейса для большого количества устройств. OpenGL ES 3.0 добавляет в OpenGL ES 2.0 поддержку многих приемов, оптимизаций и улучшений качества изображения. Следующие разделы предоставляют обзор основных возможностей, которые были добавлены в OpenGL ES 3.0. Каждая из этих возможностей позже будет детально описана.

Текстурирование

OpenGL ES 3.0 вводит новые возможности, связанные с текстурированием:

- sRGB-текстуры и фреймбуферы – позволяют приложениям осуществлять рендеринг с учетом гамма-коррекции. Текстуры могут хранить значения в пространстве sRGB и переводиться в линейное пространство при чтении в шейдере и затем обратно переводиться в sRGB при записи во фреймбуфер. Это позволяет добиться более высокого качества графики, используя вычисление освещения в линейном пространстве;
- двухмерные текстурные массивы – тип текстуры, хранящей внутри себя массив двухмерных текстур. Подобные массивы могут быть использованы, например, для анимации текстуры. До появления двухмерных текстурных массивов подобная анимация делалась путем помещения отдельных кадров анимации в одну большую текстуру и последующего изменения текстурных координат. С двухмерными текстурными массивами каждый кадр анимации может быть сохранен в своем слое массива;
- трехмерные текстуры – хотя некоторые реализации OpenGL ES 2.0 поддерживали трехмерные текстуры через расширение, OpenGL ES 3.0 сделал их поддержку обязательной. Трехмерные текстуры важны для многих медицинских приложений, таких как рендеринг воксельных данных (например, полученных при томографии);
- текстуры со значениями глубины и вычислением тени – позволяют сохранить буфер глубины в текстуре. Наиболее распространенным использованием этой возможности является расчет теней, когда строится буфер глубины для положения источника света, затем он используется для определения того, лежит ли точка в тени. Кроме поддержки таких текстур, OpenGL ES 3.0 дает возможность выполнить сравнения значений во время чтения из подобной текстуры, позволяя тем самым использовать билинейную фильтрацию для таких текстур;
- бесшовные кубические текстурные карты – в OpenGL ES 2.0 рендеринг с использованием кубических текстурных карт мог давать артефакты на границе между гранями такой карты. В OpenGL ES 3.0 чтение из кубической текстурной карты может быть организовано таким образом, что при

фильтрации будут использоваться значения с соседней грани, убирая подобные артефакты;

- текстуры с поддержкой значений с плавающей точкой – OpenGL ES 3.0 заметно расширяет список поддерживаемых форматов текстур. Поддерживаются текстуры с 16-битовыми значениями с плавающей точкой, и для них поддерживается фильтрация, также поддерживаются текстуры с 32-битовыми значениями с плавающей точкой, но для них фильтрация уже не поддерживается. Поддержка текстур со значениями с плавающей точкой важна для многих приложений, включая HDR-рендеринг и вычисления общего назначения на GPU;
- поддержка сжатия текстур ETC2/EAC – в то время как некоторые реализации OpenGL ES 2.0 предоставляли поддержки работы со сжатыми текстурами, используя специфические для разработчика форматы (например, ATC для Qualcomm, PVRTC для Imagination Technologies, Ericsson Texture Compression для Sony Ericsson), не было стандартного формата для работы со сжатыми текстурами, который могли бы использовать разработчики. В OpenGL ES 3.0 поддержка ETC2/EAC обязательна. Форматы ETC2/EAC предоставляют сжатие для RGB888, RGBA8888 и одно- и двухкомпонентных текстур со знаковыми и беззнаковыми данными. Сжатие текстур дает ряд преимуществ, включая большее быстродействие (из-за лучшей утилизации текстурного кэша) и уменьшенное потребление памяти GPU;
- текстуры с целочисленными компонентами – OpenGL ES 3.0 добавляет возможность осуществлять рендеринг и чтение из текстур, хранящих ненормализованные знаковые или беззнаковые 8/16/32-битовые целые значения;
- дополнительные форматы текстур – в дополнение к уже упомянутым форматам OpenGL ES 3.0 включает поддержку 11-11-10 RGB текстур с плавающей точкой, RGB текстур 9-9-9-5 с общей экспонентой и 10-10-10-2 целочисленных текстур и 8-битовых нормализованных текстур со знаком;
- текстуры, размеры которых не являются степенями двух (NPOT) – при создании текстур можно теперь использовать размеры, не являющиеся степенями двух. Это полезно во многих ситуациях, таких как случай, когда текстура состоит из данных с камеры или видеоряда;
- дополнительный контроль над уровнем детализации текстур – этот уровень детализации обычно используется для выбора слоя из mipmap-пирамиды, теперь может быть обрезан по заданному диапазону. Также базовый и максимальный уровни могут быть обрезаны. Все эти возможности позволяют организовывать стриминг текстур. По мере того как новые уровни поступают, изменяется базовый уровень и значение уровня детализации плавно изменяется. Это очень полезно, например, при скачивании всей пирамиды уровней по сети;
- возможность автоматической перестановки компонент текстуры;
- неизменяемые текстуры – предоставлен механизм, позволяющий приложению задать формат и размер текстуры до загрузки в нее данных. При этом

текстура становится неизменяемой, и драйвер OpenGL ES может заранее выполнять все проверки. Это может улучшить быстродействие за счет того, что драйвер может отказаться от проверок во время рендеринга;

- увеличены минимальные размеры текстур – все реализации OpenGL ES 3.0 теперь должны поддерживать заметно большие по размерам текстуры, чем OpenGL ES 2.0. Например, минимальным поддерживаемым размером двухмерной текстуры в OpenGL ES 2.0 был 64, но в OpenGL ES 3.0 был увеличен до 2048.

Шейдеры

OpenGL ES 3.0 вводит серьезные изменения в язык для написания шейдеров и ряд новых возможностей в API для поддержки новых возможностей шейдеров.

- Шейдеры в бинарном виде – в OpenGL ES 2.0 можно было сохранять шейдеры в бинарном формате, но для их объединения в программу все равно требовалось выполнить линковку. В OpenGL ES 3.0 полностью собранная программа (состоящая из вершинного и фрагментного шейдеров) может быть сохранена в бинарный формат без необходимости в последующей линковке. Это может уменьшить время загрузки приложения. Также OpenGL ES 3.0 предоставляет интерфейс, позволяющий получить бинарное представление программы из драйвера, так что для использования бинарного представления не нужны внешние утилиты.
- Обязательная поддержка компиляции во время выполнения – в OpenGL ES 2.0 поддержка компиляции шейдеров во время выполнения была необязательной. Целью было уменьшить потребление драйвером памяти, но это дорого обошлось разработчикам, так как им пришлось использовать утилиты от разработчиков GPU для получения шейдеров. В OpenGL ES 3.0 все реализации должны поддерживать компиляцию во время выполнения.
- Неквадратные матрицы – поддерживаются новые типы матриц, которые не являются квадратными, и были добавлены соответствующие вызовы в API для работы с uniform-переменными. Неквадратные матрицы могут уменьшить число команд, необходимых для осуществления преобразований. Например, при выполнении аффинного преобразования можно использовать матрицы 4×3 вместо матрицы 4×4 в случае, когда последняя строка равна (0, 0, 0, 1), тем самым уменьшая общее число требуемых команд для выполнения преобразования.
- Полная поддержка целых чисел – в ESSL 3.00 полностью поддерживаются целочисленные скалярные и векторные типы. Есть ряд встроенных функций для преобразований между числами с плавающей точкой и целыми числами, включая возможность читать целые числа из текстур и выводить целые числа в выходные буферы.
- Использование выборок centroid – для избегания артефактов при мультисэмплинге выходные переменные вершинного шейдера (и входные переменные фрагментного шейдера) могут быть описаны как centroid.

- Выбор типа интерполяции – в OpenGL ES 2.0 всегда использовалась линейная интерполяция вдоль всего примитива. В ESSL 3.00 можно явно задать тип интерполяции.
- Uniform-блоки – значения uniform-переменных могут быть сгруппированы вместе в uniform-блоки. Использование uniform-блоков более эффективно, кроме того, такие блоки могут применяться сразу несколькими шейдерными программами совместно.
- Спецификаторы размещения – входные значения для вершинного шейдера теперь могут быть описаны с использованием спецификаторов размещения, позволяющих явно задать их размещение в тексте шейдера без специальных вызовов API. Также спецификаторы размещения могут быть использованы во фрагментном шейдере для явной привязки выходных значений при рендеринге сразу в несколько текстур. Еще подобные спецификаторы могут быть использованы для управления размещением в памяти для uniform-блоков.
- Номера экземпляра и вершины – номер вершины теперь доступен в вершинном шейдере, так же как и номер экземпляра при использовании дублирования геометрии (instancing).
- Глубина фрагмента – фрагментный шейдер теперь может явно управлять значением глубины фрагмента, а не полагаться на интерполяцию.
- Новые встроенные функции – в ESSL 3.00 вводится много новых встроенных функций для поддержки работы с текстурами, производных, преобразований данных и использования 16-битовых чисел с плавающей точкой, матричных и математических операций.
- Ослабленные ограничения – ESSL 3.00 заметно ослабляет ограничения на шейдеры. Шейдеры больше не ограничены в числе команд, полностью поддерживаются циклы и ветвление на основе значений переменных, поддерживается индексирование массивов.

Геометрия

OpenGL ES 3.0 вводит несколько новых возможностей, связанных с заданием геометрии и управлением рендеринга примитивов.

- Преобразование обратной связи – позволяет выход вершинного шейдера сохранить в буферном объекте. Это полезно в целом ряде случаев, когда выполняется анимация полностью на GPU, вообще не задействуется CPU – например, анимация частиц или моделирование физики с использованием рендеринга в вершинный буфер.
- Логические проверки видимости – позволяет приложению запросить, прошел ли хоть фрагмент из последней команды или группы команд тест глубины. Эта возможность может использоваться во многих случаях, таких как определение видимости для имитации отражения от линз, оптимизация обработки объектов, чей ограничивающий прямоугольный параллелепипед невидим.

- Дублирование геометрии – позволяет эффективно выводить объекты, обладающие похожей геометрией, но отличающиеся атрибутами (такими как матрица преобразования, цвет или размер). Эта возможность полезна при выводе большого количества похожих объектов, например при рендеринге толпы.
- Сброс примитива – при использовании полос треугольников в OpenGL ES 2.0 для вывода нового примитива приложение должно было вставлять специальные индексы (соответствующие вырожденному треугольнику) в индексный буфер. В OpenGL ES 3.0 может использоваться специальное значение, обозначающее начало нового примитива. Это устраняет необходимость в использовании вырожденных треугольников при примененеии полос треугольников для вывода геометрии.
- Новые форматы вершин – в OpenGL ES 3.0 поддерживаются новые форматы вершин, включающие нормализованный и ненормализованный формат 10-10-10-2, 8/16/32-битовые целочисленные и 16-битовые атрибуты с плавающей точкой.

Буферные объекты

OpenGL ES 3.0 вводит много новых буферных объектов для увеличения эффективности и гибкости задания данных в различных местах графического конвейера.

- Uniform-буферы – предоставляют эффективный метод для хранения и привязывания больших блоков uniform-значений. Эти буферы могут быть использованы для уменьшения цены передачи uniform-значений в шейдеры, что часто являлось узким местом в OpenGL ES 2.0.
- Объекты для хранения состояния вершинных массивов. Фактически являются контейнерами для состояния вершинных массивов. Их использование позволяет приложению переключить состояние всего за один вызов вместо необходимости использовать группу вызовов.
- Сэмплеры – отделяют способ выборки из текстуры (режим отсечения текстурных координат и способ фильтрации) от самого текстурного объекта. Это позволяет совместное использование различных способов выборки.
- Объекты синхронизации – предоставляют механизм, позволяющий приложению проверить, закончил ли выполнение на GPU определенный набор команд OpenGL ES. Близким новым понятием является барьер, позволяющий приложению сообщить GPU, что он должен дождаться, пока заданный набор операций OpenGL ES завершит выполнение, прежде чем ставить в очередь новые команды.
- Пиксельные буферы – позволяют приложению выполнять асинхронное копирование данных. В основном ориентировано на предоставление быстрого способа копирования данных между CPU и GPU, в то время как приложение может продолжать работать.
- Отображение части буфера – позволяет приложению отобразить область буфера для доступа со стороны CPU. Это может дать более высокое быстродействие, по сравнению с обычным отображением всего буфера.

- Копирование между буферами – предоставляет механизм для эффективного копирования данных из одного буфера в другой без вмешательства CPU.

Фреймбуфер

OpenGL ES 3.0 вводит много новых возможностей по внеэкранному рендерингу с использованием фреймбуферов:

- рендеринг сразу в несколько текстур (MRT) – позволяет приложению осуществлять рендеринг сразу в несколько цветовых буферов. В этом случае фрагментный шейдер выдает сразу несколько цветов, по одному на каждый цветовой буфер. Используется в ряде методов рендеринга, например при отложенном освещении;
- рендербуферы с поддержкой мультисэмплинга – позволяют приложению осуществлять рендеринг во внеэкранный буфер с поддержкой мультисэмплинга. Такие рендербуферы не могут быть непосредственно привязаны к текстурам, но для них можно выполнить специальную операцию копирования с использованием одного сэмпла;
- подсказки о необходимости использования фреймбуфера – многие реализации OpenGL ES 3.0 основаны на GPU, использующем тайловый рендеринг (разбирается в соответствующей части главы 12). Часто такой подход приводит к значительным затратам в случае, когда необходимо восстановить содержимое тайлов для дальнейшего рендеринга во фреймбуфер. Данный механизм позволяет сообщить драйверу, что содержимое фреймбуфера больше не нужно. Это позволяет драйверам выполнять различные оптимизации, в частности пропускать восстановление тайлов. Подобная функциональность крайне важна для получения высокого быстродействия в ряде приложений, особенно в случае большого объема внеэкранного рендеринга;
- новые уравнения для смешивания цветов – в качестве уравнений для смешивания цветов в OpenGL ES 3.0 поддерживаются функции min/max.

OpenGL ES 3.0 и обратная совместимость

OpenGL ES 3.0 обратно совместим с OpenGL ES 2.0. Это значит, что практически любое приложение, написанное с использованием OpenGL ES 2.0, будет работать на реализации OpenGL ES 3.0. Однако есть некоторые незначительные изменения в последнем, которые затронут небольшое число приложений в смысле обратной совместимости. Так, объекты фреймбуфера больше не разделяются между контекстами, для кубических текстурных карт всегда используется бесшовная фильтрация, и есть небольшие изменения в том, как знаковые числа с фиксированной точкой преобразуются в числа с плавающей точкой.

Тот факт, что OpenGL ES 3.0 обратно совместим с OpenGL ES 2.0, отличается от того, что было сделано по отношению к совместимости OpenGL ES 2.0 с предыдущей версией OpenGL ES. OpenGL ES 2.0 не является обратно совмес-

тимым с OpenGL ES 1.x. OpenGL ES 2.0/3.0 не поддерживает фиксированный конвейер рендеринга, который поддерживает OpenGL ES 1.x. Вершинные шейдеры в OpenGL ES 2.0/3.0 заменяют блок обработки вершин в OpenGL ES 1.x. В фиксированном конвейере вершинный блок реализует заданные преобразования вершин и вычисление освещения, преобразование или генерацию текстурных координат и вычисление цвета в вершине. Аналогично фрагментный шейдер заменяет блок наложения текстуры из фиксированного конвейера в OpenGL ES 1.x. Блоки наложения текстур из фиксированного конвейера реализуют наложение текстур на каждом шаге. Цвет текстуры накладывается на диффузный цвет и результат предыдущего текстурного блока при помощи заданного набора операций, таких как сложение, умножение, вычитание и скалярное произведение.

Рабочая группа по OpenGL ES 2.0/3.0 выступила против обратной совместимости с OpenGL ES 1.x по следующим причинам:

- поддержка фиксированного конвейера в OpenGL ES 2.0/3.0 означает, что API должен поддерживать более одного способа реализации возможностей, нарушая тем самым один из принципов, используемых при определении набора поддерживаемых возможностей. Программируемый конвейер позволяет приложениям реализовать фиксированный конвейер при помощи шейдеров, поэтому нет никакого смысла поддерживать обратную совместимость с OpenGL ES 1.x;
- согласно откликам от разработчиков, в большинстве игр программируемый конвейер и фиксированный конвейер не смешивались. Это значит, что игра пишется целиком либо на программируемом конвейере, либо на фиксированном конвейере. Если вы используете программируемый конвейер, то нет никакого смысла использовать фиксированный конвейер, поскольку в вашем распоряжении гораздо больше гибкости;
- размер драйвера OpenGL ES 2.0/3.0 будет гораздо больше, если нужно будет поддерживать и фиксированный конвейер, и программируемый конвейер. Для устройств, на которые нацелен OpenGL ES, объем памяти играет важную роль. Разделение между фиксированным конвейером в OpenGL ES 1.x и программируемым конвейером в OpenGL ES 2.0/3.0 означает, что разработчикам больше не нужно включать поддержку OpenGL ES 1.x в драйвер.

EGL

Для выполнения команд OpenGL ES нужны контекст и поверхность для рендеринга. Контекст хранит в себе соответствующее состояние OpenGL ES. Поверхность для рисования – эта та поверхность, на которую будут выведены примитивы. Поверхность для вывода определяет, какие типы буферов требуются для рендеринга, такие как буфер цвета, буфер глубины и буфер трафарета. Поверхность для вывода также определяет размер в битах всех требуемых буферов.

OpenGL ES API не упоминает, как контекст для рендеринга создается или как этот контекст привязывается к используемой оконной системе. EGL – это вариант интерфейса между API для рендеринга от Khronos, такими как OpenGL ES, и оконной системой; на самом деле не требуется предоставлять EGL при реализации OpenGL ES. Разработчики должны обратиться к документации по своей платформе, для того чтобы определить поддерживаемый интерфейс. На момент написания единственной платформой, поддерживающей OpenGL ES и не поддерживающей EGL, является iOS.

Любое приложение на OpenGL ES, перед тем как можно осуществлять рендеринг, должно выполнить следующие действия при помощи EGL:

- узнать, какие дисплеи доступны на устройстве, и инициализировать их. Например, смартфон-раскладушка может иметь два LCD-дисплея, и мы можем захотеть использовать OpenGL ES для рендеринга на поверхность, которая может быть видна на любом из них или сразу на обоих;
- создать поверхность для рендеринга. Поверхности, созданные при помощи EGL, можно разделить на видимые поверхности и внеэкранные поверхности. Видимые поверхности подключаются к оконной системе, в то время как внеэкранные поверхности – это буферы пикселов, которые не отображаются, но могут быть использованы для рендеринга в них. Эти поверхности могут быть использованы для рендеринга в текстуру, и их можно совместно использовать между несколькими API от Khronos;
- создать контекст для рендеринга. Для создания контекста для рендеринга OpenGL ES требуется EGL. Этот контекст должен быть присоединен к соответствующей поверхности перед началом рендеринга.

EGL API реализует описанные возможности, так же как и дополнительные возможности, такие как управление потреблением энергии, поддержка сразу нескольких контекстов, совместное использование объектов (таких как текстуры или вершинные буферы) разными контекстами и механизм для получения указателей на функции, предоставляемые расширениями EGL и OpenGL ES для данной реализации.

Последней версией EGL является 1.4.

Программирование с OpenGL ES 3.0

Для написания приложения с использованием OpenGL ES 3.0 вам нужно знать, какие заголовочные файлы необходимо включить и с какими библиотеками нужно собирать ваше приложение. Также полезно понять синтаксис, используемый командами и параметрами EGL и OpenGL.

Библиотеки и заголовочные файлы

Приложения для OpenGL ES 3.0 должны быть собраны с использованием следующих библиотек: библиотека OpenGL ES 3.0 с названием libGLESv2.lib и библиотека EGL с названием libEGL.lib.

Приложения, использующие OpenGL ES 3.0, также должны включить соответствующие заголовочные файлы для OpenGL ES 3.0 и EGL. Следующие заголовочные файлы должны быть включены любым приложением на OpenGL ES 3.0:

```
#include <EGL/egl.h>
#include <GLES3/gl3.h>
```

Файл `egl.h` – это заголовочный файл библиотеки EGL, а файл `gl3.h` – это заголовочный файл библиотеки OpenGL ES 3.0. Также приложение может включать необязательный файл `gl2ext.h`, содержащий список одобренных Khronos расширений для OpenGL ES 2.0/3.0.

Заголовочные файлы и имена библиотек зависят от используемой платформы. Рабочая группа по OpenGL ES пыталась определить имена заголовочных и библиотечных файлов и как они должны быть организованы, но это может не соблюдаться на некоторых платформах. Разработчики должны использовать документацию для своей платформы, для того чтобы узнать, как названы и как организованы заголовочные файлы и файлы библиотек. Официальные заголовочные файлы OpenGL ES поддерживаются Khronos и доступны по адресу **http://khronos.org/registry/gles**. Исходный код примеров к данной книге также включает в себя копию заголовочных файлов (работающих с исходным кодом, описанным в следующей главе).

Синтаксис EGL

Все команды EGL начинаются с префикса `egl` и используют заглавную букву для начала каждого слова, образующего имя команды (например, `eglCreateWindowSurface`). Аналогично все типы данных в EGL начинаются с префикса `Egl` и используют заглавную букву для каждого слова, входящего в имя типа, за исключением `EGLint` и `EGLenum`.

Таблица 1.1 кратко описывает используемые типы EGL.

Таблица 1.1. Типы данных в EGL

Тип данных	Тип в C	Тип в EGL
32-битовое целое число	`int`	`EGLint`
32-битовое беззнаковое целое число	`unsigned int`	`EGLBoolean`, `EGLenum`
Указатель	`void *`	`EGLConfig`, `EGLContext`, `EGLDisplay`, `EGLSurface`, `EGLClientBuffer`

Синтаксис команд OpenGL ES

Все команды OpenGL ES начинаются с префикса `gl` и используют заглавную букву для каждого слова, входящего в название команды (например, `glBlendEquation`). Аналогично типы данных OpenGL ES начинаются с префикса `GL`.

Кроме того, некоторые команды могут получать аргументы различными способами. Эти способы, или типы, включают в себя число аргументов (от одного до четырех аргументов), тип данных, используемый для аргументов (byte [b], unsigned byte [ub], short [s], unsigned short [us], int [i], float [f]), и переданы ли аргументы как вектор (v). Дальше идет несколько иллюстрирующих это примеров.

Следующие две команды эквивалентны, за исключением того, что одна задает значение uniform-переменной как число с плавающей точкой, а другая – как целое число.

```
glUniform2f(location, 1.0f, 0.0f);
glUniform2i(location, 1, 0)
```

Следующие строки также содержат эквивалентные команды, за исключением того, что одна из них передает аргументы как вектор, а другая – нет.

```
GLfloat coord[4] = { 1.0f, 0.75f, 0.25f, 0.0f };
glUniform4fv(location, coord);
glUniform4f(location,
            coord[0], coord[1], coord[2], coord[3]);
```

Таблица 1.2 описывает суффиксы команд и типы аргументов, используемых в OpenGL ES.

Таблица 1.2. Суффиксы команд OpenGL ES и соответствующие им типы данных

Суффикс	Тип данных	Тип в C	Тип в GL
b	8-битовое целое со знаком	signed char	GLbyte
ub	8-битовое целое без знака	unsigned char	GLubyte, GLboolean
s	16-битовое целое со знаком	short	GLshort
us	16-битовое целое без знака	unsigned short	GLushort
i	32-битовое целое со знаком	int	GLint
ui	32-битовое целое без знака	unsigned int	GLuint, GLbitfield, GLenum
x	Число с фиксированной точкой в формате 16.16	int	GLfixed
f	32-битовое число с плавающей точкой	float	GLfloat, GLclampf
i64	64-битовое целое число со знаком	khronos_int_64_t (зависит от платформы)	GLint64
ui64	64-битовое целое число без знака	khronos_uint_64_t (зависит от платформы)	GLuint64

И еще OpenGL ES определяет тип GLvoid. Этот тип используется для команд OpenGL ES, принимающих на вход указатели.