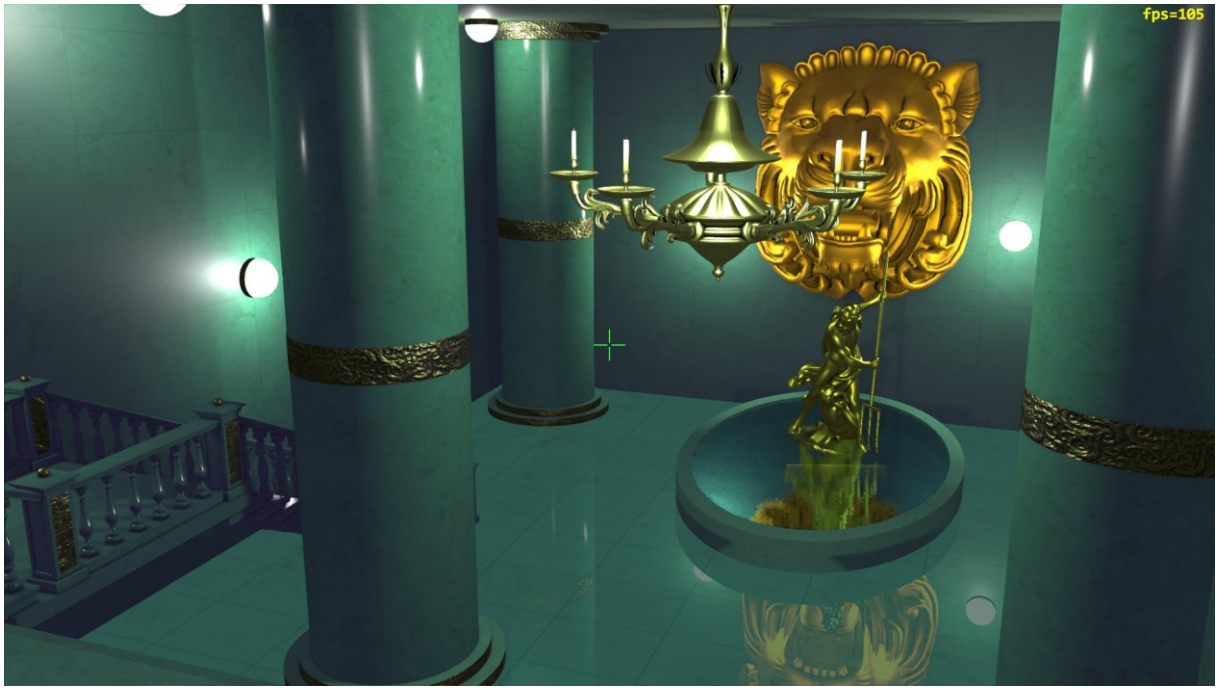
Владимирский государственный университет

С. А. САМОЙЛОВ

ПРОГРАММИРОВАНИЕ ТРЕХМЕРНОЙ ГРАФИКИ СРЕДСТВАМИ БИБЛИОТЕКИ DIRECTX 11

Учебное пособие



Владимир 2025

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»

С. А. САМОЙЛОВ

ПРОГРАММИРОВАНИЕ ТРЕХМЕРНОЙ ГРАФИКИ СРЕДСТВАМИ БИБЛИОТЕКИ DIRECTX 11

Учебное пособие

Электронное издание

Владимир 2025

ISBN 978-5-9984-2020-7

УДК 004.94
ББК 32.972.13

Рецензенты:
Доктор технических наук, доцент
зав. кафедрой радиотехники Национального исследовательского
Нижегородского государственного университета
им. Н. И. Лобачевского
Е. С. Фитасов

Кандидат физико-математических наук, доцент
доцент кафедры функционального анализа и его приложений
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых
А. Е. Додонов

Издается по решению редакционно-издательского совета ВлГУ

Самойлов, С. А.

Программирование трехмерной графики средствами библиотеки DirectX 11 [Электронный ресурс] : учеб. пособие / С. А. Самойлов ; Владим. гос. ун-т им. А. Г. и Н. Г. Столетовых. – Владимир : Изд-во ВлГУ, 2025. – 199 с. – ISBN 978-5-9984-2020-7. – Электрон. дан. (5,40 Мб). – 1 электрон. опт. диск (CD-ROM). – Систем. требования : Intel от 1,3 ГГц ; Windows XP/7/8/10 ; Adobe Acrobat Reader ; дисковод CD-ROM. – Загл. с титул. экрана.

Рассмотрены основные понятия и определения трехмерной графики. Описаны основные методы отображения трехмерных сцен средствами DitectX 11. Приведены модели физически корректного освещения. Представлены примеры вершинных и фрагментных шейдеров.

Предназначено для студентов направления подготовки 02.03.01 «Математика и компьютерные науки» очной формы обучения, будет полезно студентам, специализирующимся на программировании в области трехмерной графики.

Рекомендовано для формирования профессиональных компетенций в соответствии с ФГОС ВО.

Ил. 86. Библиогр.: 31 назв.

ISBN 978-5-9984-2020-7

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Развитие средств вычислительной техники и создание графических сопроцессоров – аналогов будущих графических ускорителей в конце 80-х – начале 90-х годов XX века привело к появлению нового направления в программировании систем реального времени – программирования компьютерной графики. А создание относительно мощных процессоров в середине 90-х годов XX века привело к появлению систем виртуальной реальности, основанных на трехмерной графике. Особую популярность данный вид компьютерных программ завоевал за счет эффекта присутствия и погружения в виртуальный трехмерный мир.

Возникновение графических ускорителей с фиксированным конвейером рендеринга, способных выполнять простейшие геометрические и алгебраические операции с большой скоростью и в большом количестве, дало толчок к созданию программ, стремящихся к отображению трехмерной графики с высоким качеством и в режиме реального времени. Первым графическим ускорителем с графическим процессором (GPU), появившимся в 1999 году, следует считать видеокарту GeForce-256 от фирмы Nvidia. Эта видеокарта интегрировала механизмы трансформации, освещения, растеризации, текстурирования и рендеринга в отличие от других видеокарт того времени, которые часть задач возлагали на центральный процессор CPU.

В 2006 году Nvidia представила видеокарту GeForce 8800 GTX, которая содержала не отдельные процессоры для вершинных и пиксельных шейдеров, а процессоры общего назначения, т. е. появилась возможность динамического перераспределения ресурсов в зависимости от задачи и рабочей нагрузки. Важный этап в развитии графических укорителей – появление вычислений общего назначения на графических процессорах (GPGPU). Параллельная обработка данных с высокой скоростью и в большом количестве дала возможность физически корректной визуализации трехмерных сцен.

На сегодняшний день наиболее передовой и реалистичной считается технология визуализации трехмерных сцен за счет трассировки

лучей (Ray tracing), что позволяет осуществлять освещение, затенение и отражение в трехмерных сценах с наиболее высокой степенью реализма по сравнению с другими алгоритмами рендеринга. Наряду с развитием графических ускорителей и графических процессоров активно развивались и различные программные интерфейсы (API Application Programming Interface) для взаимодействия с ними.

На сегодняшний день наиболее популярные графические API – DirectX, OpenGL и Vulkan. Первой из этой тройки появился API OpenGL (Open Graphics Library) в 1992 году. OpenGL – это аппаратно независимый интерфейс, который включает в себя более трехсот функций для отображения сложных трехмерных сцен. Архитектура OpenGL ориентируется на скрытие различий между аппаратными платформами и скрытие сложности адаптации различных графических ускорителей, предоставляя единый API, в том числе за счет программной эмуляции отсутствующих аппаратных средств.

API Vulkan впервые был представлен в 2015 году Khronos Group. Vulkan API изначально был анонсирован как новое поколение OpenGL. Vulkan также кроссплатформенный API, призванный отображать приложения с трехмерной графикой с более высокой производительностью и меньшей нагрузкой на процессор.

API DirectX впервые появился в 1995 году вместе с Windows-95. Изначально нацеленный на разработку видеоигр, DirectX завоевал популярность и в инженерно-математическом программном обеспечении. DirectX предоставляет разработчику прямой доступ к видеокарте и другим аппаратным частям системы, в то время как Windows предоставляет весьма ограниченный доступ. На текущий момент существует версия DirectX 11.3, поддерживаемая операционной системой Windows-10, и DirectX 12, поддерживаемая Windows-10/11.

В настоящем учебном пособии излагаются основные сведения о графическом API DirectX 11.3. Издание рассчитано на читателя, имеющего базовые знания в области программирования на языке C++ и знакомого с его основными понятиями, определениями и синтаксисом.

В первой главе изложены основные сведения о принципах разработки современных приложений с использованием трехмерной графики и дано определение графического конвейера. Показан основной матричный математический аппарат, дано определение позиции,

нормали и текстурных координат, а также приведен принцип построения буфера глубины.

Во второй главе рассказано об инициализации объектов и интерфейсов библиотеки DirectX, создании вершинных, индексных и константных буферов. Описаны средства управления стадиями графического конвейера. Показаны примеры создания вершинных, фрагментных и геометрических шейдеров. Приведены примеры программ для отображения трехмерных, в том числе и прозрачных, объектов.

Третья глава посвящена рассмотрению различных моделей освещения и источников света. Объясняются принципы физически корректного рендеринга, затухания света. Показаны методы построения моделей материалов, карт нормалей и карт окружения.

В четвертой главе изложен метод отложенного освещения, рассмотрены достоинства и недостатки этого метода и приведены примеры расчета освещения множественных источников света. Параллельно с методом отложенного освещения приведены методы построения теней, в том числе затенения окружением и применения теневых карт. Показан алгоритм отображения полупрозрачных объектов при отложенном освещении, а также этап финальной обработки в виде алгоритмов экранного сглаживания.

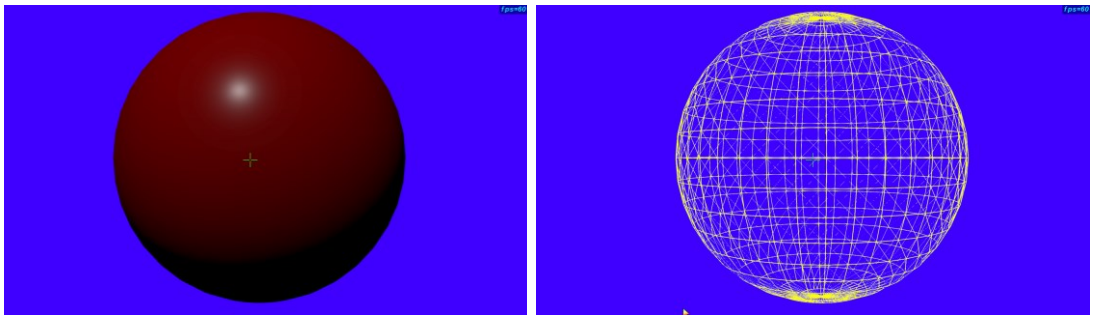
В приложениях приведены примеры реализации некоторых методов, алгоритмов и шейдеров, упомянутых на страницах учебного пособия. На сайте https://github.com/samoyow1973 можно скачать исходные коды и примеры программ, которые иллюстрируют изложенные в пособии алгоритмы и методы визуализации трехмерных сцен.

Автор выражает глубокую благодарность доктору технических наук, доценту заведующему кафедрой радиотехники Национального исследовательского Нижегородского государственного университета им. Н. И. Лобачевского *Е. С. Фитасову* и кандидату физико-математических наук доценту кафедры функционального анализа и его приложений Владимирского государственного университета имени А. Г. и Н. Г. Столетовых *А. Е. Додонову* за ценные советы по улучшению содержания учебного пособия.

Глава 1. ОСНОВНЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ ТРЕХМЕРНОЙ ГРАФИКИ

1.1. Полигоны, вершины, треугольники

Библиотека DirectX 11 способна отображать два вида графических примитивов – линии и треугольники. Таким образом, любые трехмерные объекты представляют собой набор треугольников или линий. На рис. 1.1, *а* изображен шар, состоящий из 1600 треугольников, а на рис. 1.1, *б* – каркас этого же шара.



а) *б)*

Рис. 1.1. Трехмерное изображение шара

На рис. 1.2 представлен увеличенный фрагмент поверхности шара с выделением треугольников, из которых и состоит шар.

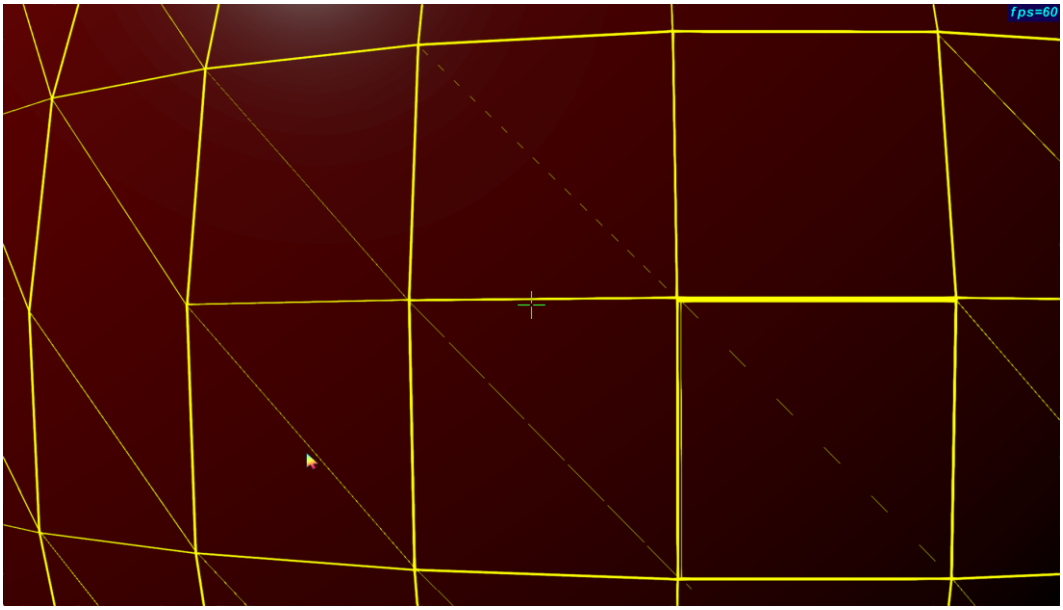


Рис. 1.2. Поверхность шара при увеличении

В трехмерной графике выбор такого примитива, как треугольник, обусловлен тем, что эта фигура содержит минимальное количество вершин, а все точки треугольника лежат всегда в одной плоскости независимо от того, какие были выбраны координаты вершин. Последняя особенность позволяет использовать простейшие формулы интерполяции при выводе треугольника на экран, что легко реализуемо на аппаратном уровне.

Таким образом, шар, изображенный на рис. 1.1, *а* на самом деле является многогранником, содержащим 860 вершин. За счет выбора большого количества треугольников, а также за счет дополнительных графических техник (речь о которых пойдет ниже) достигается визуальный эффект гладкой сферической поверхности. При уменьшении количества треугольников и вершин этот эффект будет нивелироваться, как видно на рис. 1.3.

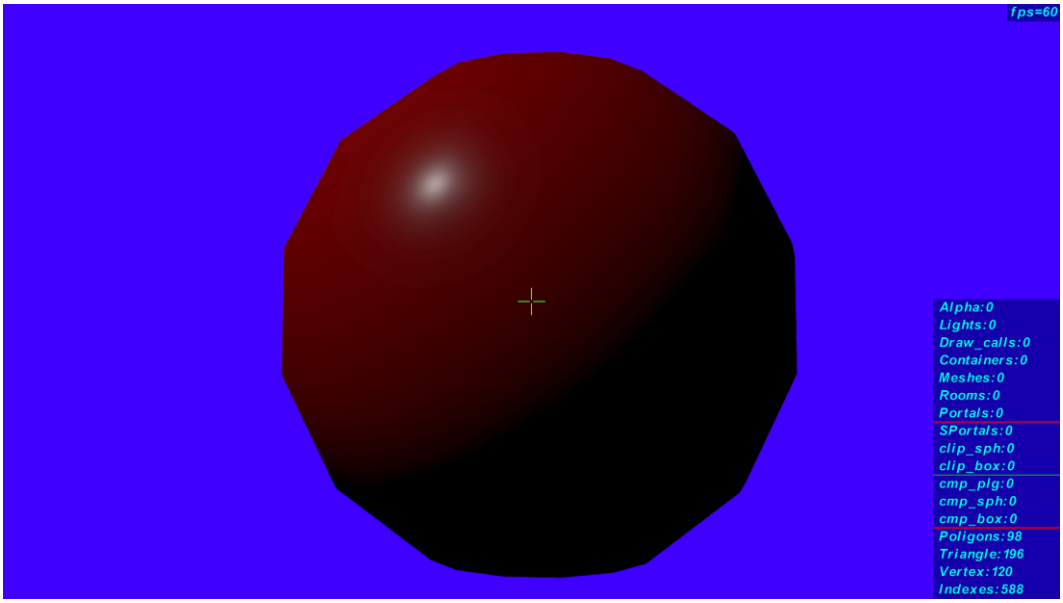


Рис. 1.3. Шар из 196 треугольников и 120 вершин

Необходимо отметить, что количество вершин уступает количеству треугольников, из которых состоит шар. Это обусловлено тем, что разные треугольники могут содержать общие вершины. Причем каждая вершина кроме трехмерных пространственных (или двумерных экранных) координат может содержать достаточно большой набор и других атрибутов, к которым можно отнести нормаль, текстурные координаты, бинормаль, тангенту, цвет, индексы костей при анимации и многое другое.

Выбор необходимых атрибутов вершин зависит от техники отображения и визуальных эффектов, которые необходимо получить. На рис. 1.4 выделены один треугольник и одна его вершина, а также показаны атрибуты этой вершины. В данном примере атрибут цвета вершины отсутствует и все вершины рисуются красным цветом. Стадия обработки вершин, а также стадия обработки пикселов на экране после интерполяции атрибутов вершин будут изложены в следующих параграфах.

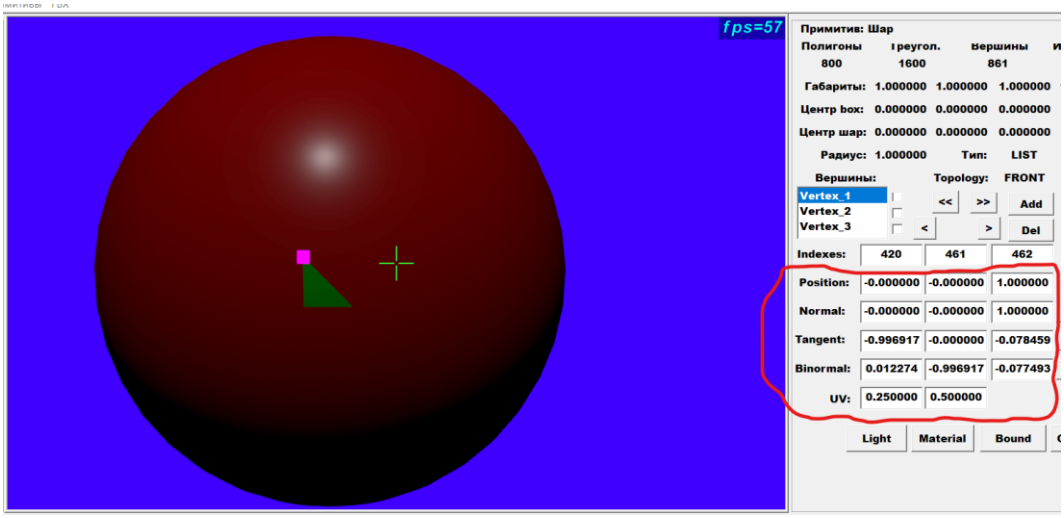


Рис. 1.4. Атрибуты вершины

При отображении трехмерного объекта необходимо передать видеокарте весь набор треугольников или линий, из которых состоит объект. Конечно, сами треугольники или линии никуда не передаются, а передаются только их вершины.

В случае отображения объекта из треугольников, как на рис. 1.1, *а*, видеокарте, или, правильнее говоря, графическому конвейеру, необходимо передать по три вершины для каждого треугольника, а в случае отображения линиями, как на рис. 1.1, *б*, – по две вершины.

При отображении треугольника на экран происходят обработка атрибутов всех вершин треугольника и дальнейшая интерполяция этих значений для каждой точки, принадлежащей треугольнику, в зависимости от удаленности от каждой из вершин. Процесс интерполяции исключительно аппаратный, следовательно, происходит с максимальной скоростью, что дает значительные преимущества по быстродействию.

На рис. 1.5 показана интерполяция цветов вершин, когда три вершины треугольника имеют красный, синий и зеленый цвета, а все точки, принадлежащие треугольнику, представляют их интерполяцию в зависимости от удаленности от той или иной вершины. Поэтому ближе к центру треугольника можно наблюдать смешение цветов.

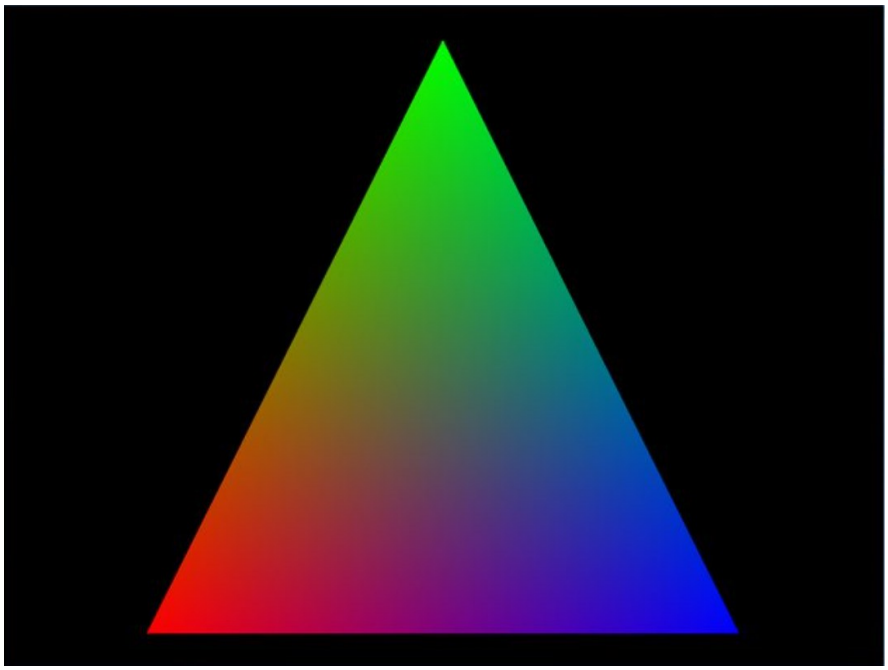


Рис. 1.5. Интерполяция разных цветов вершин треугольника

Плоские геометрические фигуры, например, трапецию, изображенную на рис. 1.6, принято называть *полигонами.* Такие графические объекты содержат несколько треугольников, лежащих в одной плоскости и обладающих общими вершинами. Так, трапеция на рис. 1.6 содержит два треугольника и четыре вершины.

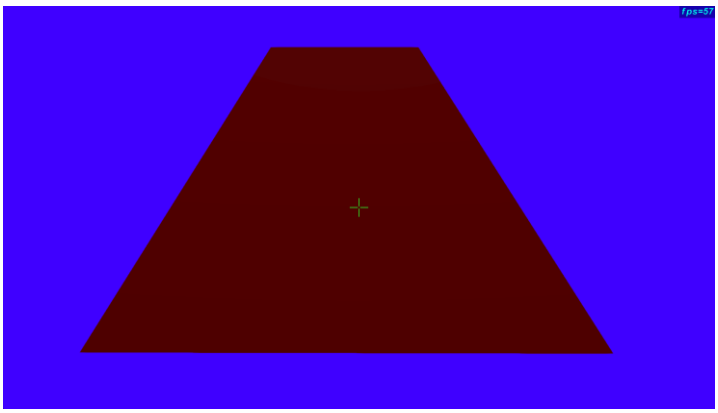


Рис. 1.6. Полигон-трапеция

Итак, каждый трехмерный объект содержит набор треугольников. Каждый треугольник из этого набора содержит три вершины, причем некоторые вершины могут быть общими для разных треугольников. Каждая вершина содержит набор атрибутов для дальнейшей обработки с использованием стадий графического конвейера и различных графических техник.

1.2. Нормали вершин

Помимо трехмерных координат в пространстве каждая вершина может еще обладать таким атрибутом, как нормаль. **Нормаль** – это нормализованный трехмерный вектор – перпендикуляр к плоскости, являющейся касательной к поверхности трехмерного объекта. Наличие нормалей позволяет использовать различные модели освещения, речь о которых пойдет в следующих параграфах.

Поскольку трехмерный объект состоит из набора треугольников, кажется, все вершины треугольника должны обладать одинаковыми нормалями, совпадающими с перпендикуляром к плоскости треугольника. На самом деле это не так. Если рассмотреть такой трехмерный объект, как шар, то можно заметить, что нормали каждой вершины совпадают по направлению с вектором, проведенным из центра шара к вершине, как показано на рис. 1.7.

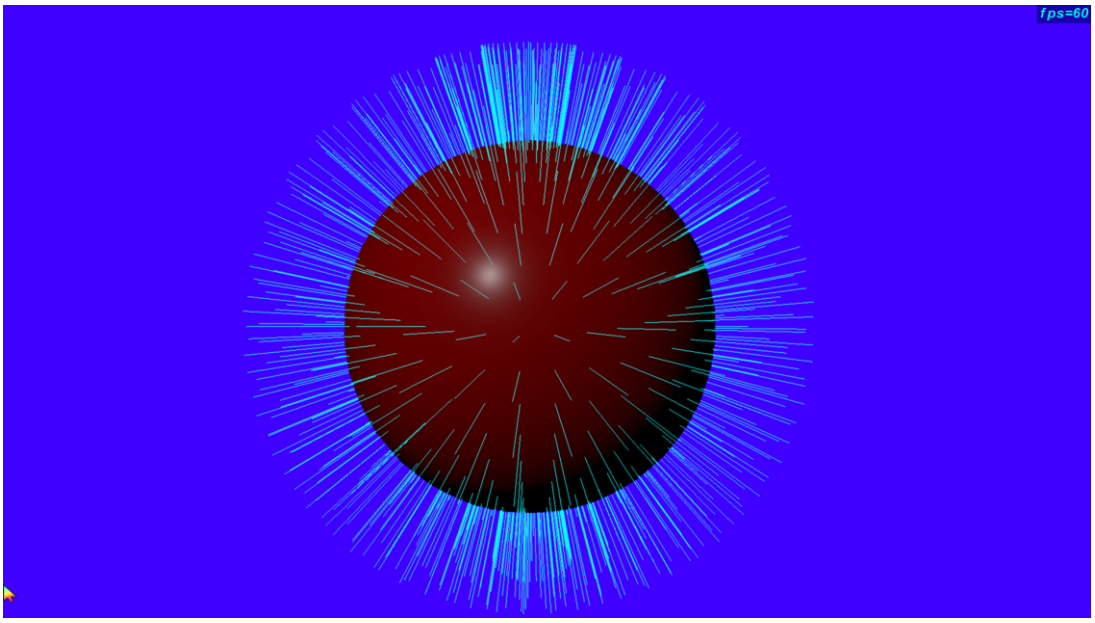


Рис. 1.7. Нормали вершин

Таким образом, каждая вершина треугольника в трехмерном объекте может обладать собственной нормалью, не совпадающей с перпендикуляром к плоскости треугольника, как показано на рис. 1.8. При выводе треугольника на экран векторы нормалей вершин интерполируются для каждой точки треугольника, что позволяет осуществлять освещение трехмерных объектов, близкое к реалистичному освещению.

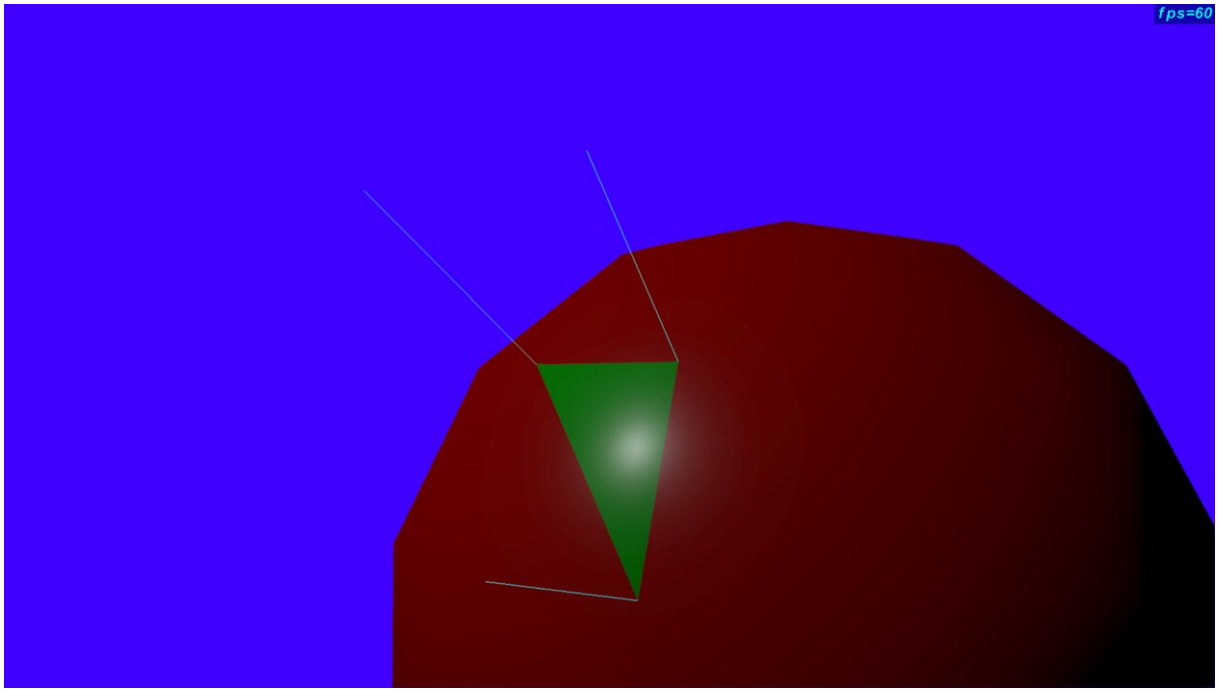


Рис. 1.8. Нормали треугольника трехмерного объекта

Однако не следует воспринимать нормали вершин как векторы, проведенные из гипотетического центра трехмерного объекта к вершине. Например, такой трехмерный объект, как куб, обладает восемью вершинами, но нормали вершин треугольника, образующих грань куба, должны совпадать с нормалью грани. Таким образом, совпадение пространственных координат вершин трехмерного объекта не означает совпадение их нормалей. В случае куба проще представить этот трехмерный объект состоящим из 6 полигонов (по числу граней), 12 треугольников и 24 вершин, каждая из которых обладает собственной нормалью, как показано на рис. 1.9.

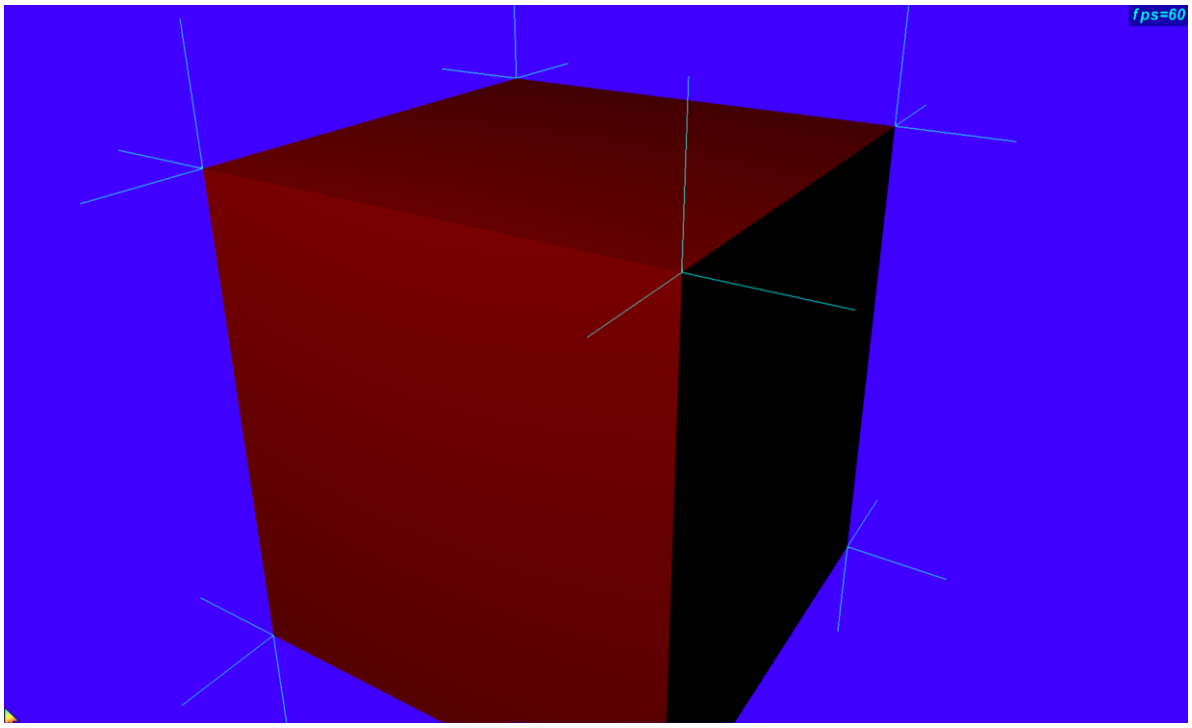


Рис. 1.9. Нормали куба

При попытке сократить количество вершин до восьми и задании общей нормали для каждой из вершин освещение трехмерного объекта перестает быть реалистичным и выглядит как на рис. 1.10.

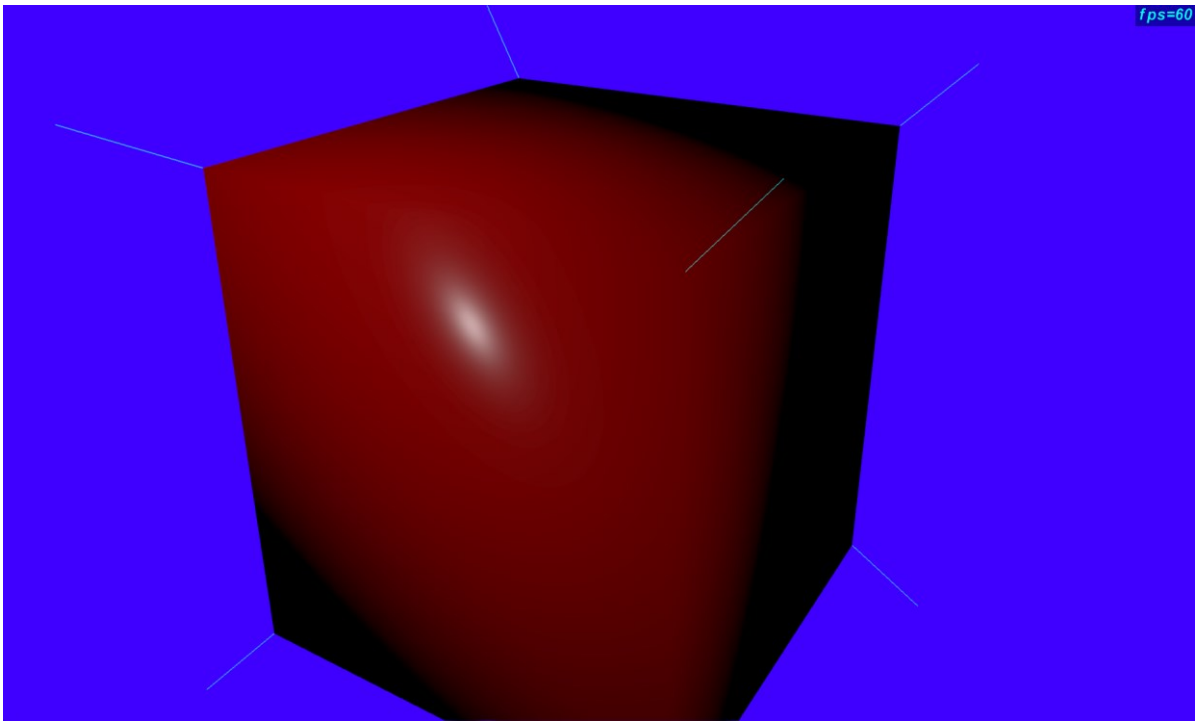


Рис. 1.10. Неверное задание нормалей вершин

1.3. Текстурирование и текстурные координаты

Пример наложения текстуры на трехмерный объект приведен на рис. 1.11. Несомненно, текстурирование – один из важных аспектов трехмерной графики, позволяющий не только получать реалистичные объекты, но и использовать различные приемы для получения фотореалистичных изображений. Возможные техники применения текстур будут рассмотрены в следующих параграфах, а пока остановимся на таких атрибутах вершин треугольников, как текстурные координаты.

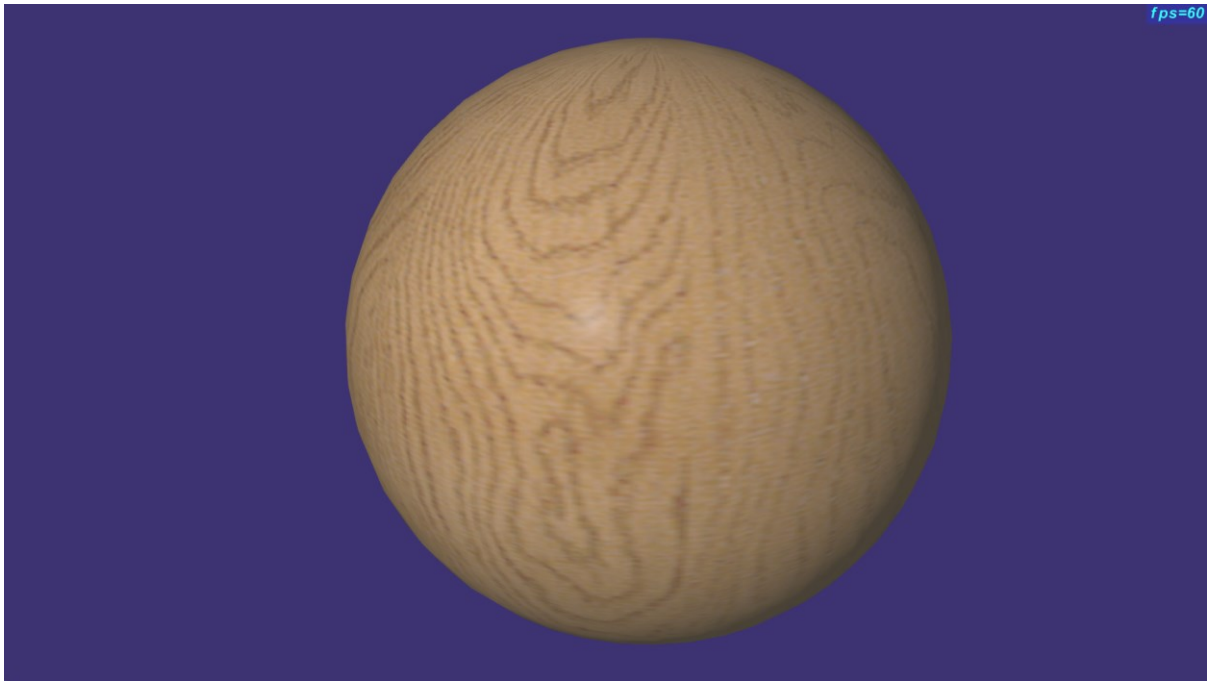


Рис. 1.11. Наложение текстуры на объект

Для наложения текстуры на объект требуется картинка текстуры, как, например, на рис. 1.12. Поскольку картинка двумерная, то можно представить ее как четырехугольник с координатами углов по двум осям от 0 до 1. Таким образом, текстурные координаты вершины представляют собой точку на картинке текстуры. При отображении треугольника на экран происходит интерполяция текстурных координат для каждого пиксела отображаемого треугольника. Полученные значения позволяют вычислить необходимый пиксел, или, правильнее говоря, тексел текстуры, и нарисовать его на экране.

Рис. 1.12. Текстура

Однако возникает ряд сложностей. Во-первых, не всегда удается точно вычислить тексел по текстурным координатам. Возможны ситуации, когда координаты указывают не точно на центр тексела, т. е. необходимо усреднение окружающих текселов. Поэтому при текстурировании используют фильтрацию, когда берут значение не конкретного тексела текстуры, а совокупности окружающих текселов. Во-вторых, если объект расположен достаточно далеко от наблюдателя, то текселы для соседних пикселов треугольника также будут достаточно далеко расположены друг от друга. Это вызывает эффект мерцания при приближении или удалении от объекта. Для того чтобы справиться с этими артефактами изображения, применяют не только саму текстуру, но и вдвое уменьшенные ее копии (mip уровни текстуры), вплоть до размера самой маленькой текстуры 1 на 1 тексел. В этом случае сопоставляют размер треугольника на экране, выбирают наиболее близкие ему по размеру mip уровни текстуры и фильтрацию производят еще и между ближайшими mip уровнями.

На рис. 1.13 приведены примеры задания различных атрибутов текстурных координат вершин для получения разных эффектов наложения текстур. Следует обратить внимание на то, что задание координат текстур более единицы может приводить к эффекту размножения текстуры.

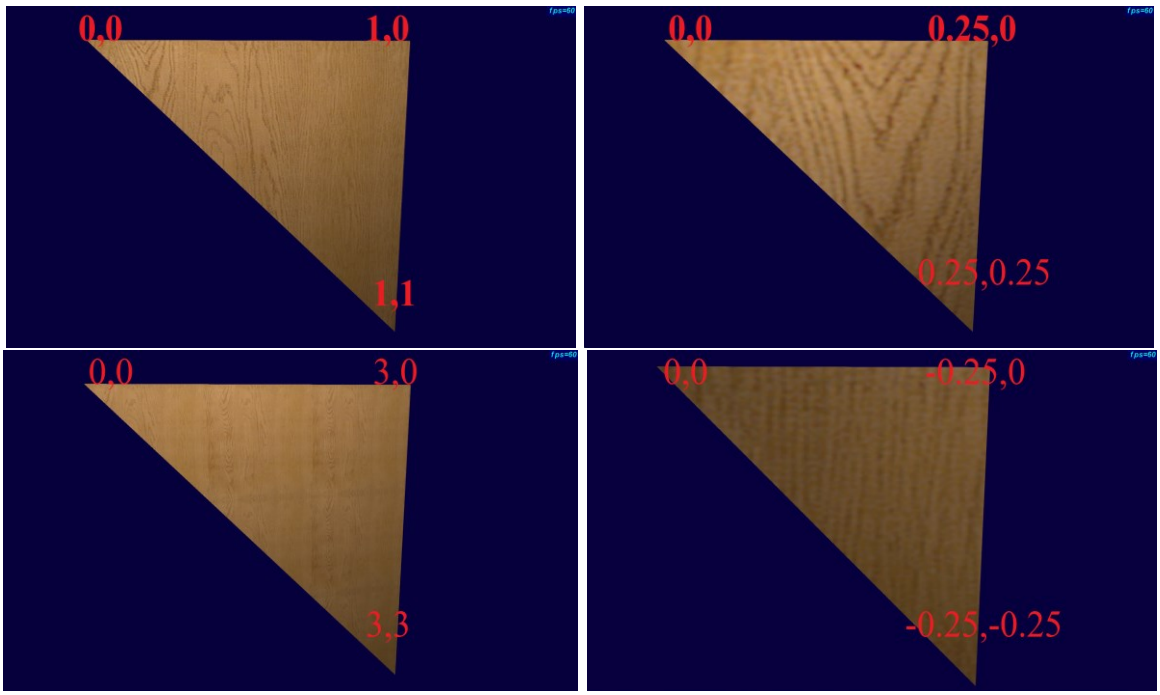


Рис. 1.13. Примеры задания текстурных атрибутов вершин

1.4. Графический конвейер

Графический конвейер, или конвейер рендеринга, предназначен для создания трехмерной графики, например графики для игр, в реальном режиме времени. Данные поступают из памяти, проходят все настраиваемые и программируемые стадии и передаются на устройство вывода (например, на экран монитора).

Все этапы графического конвейера настраиваются с помощью библиотеки DirectX11. Некоторые стадии графического конвейера являются программируемыми с помощью языка программирования шейдеров HLSL, что делает графический конвейер гибким и адаптируемым.

На рис. 1.14 представлена блок-схема упрощенного графического конвейера. Графический конвейер состоит из нескольких стадий, причем некоторые стадии имеют возможность программирования. Ресурсы памяти включают в себя различные буферы, в том числе буферы констант и текстуры.

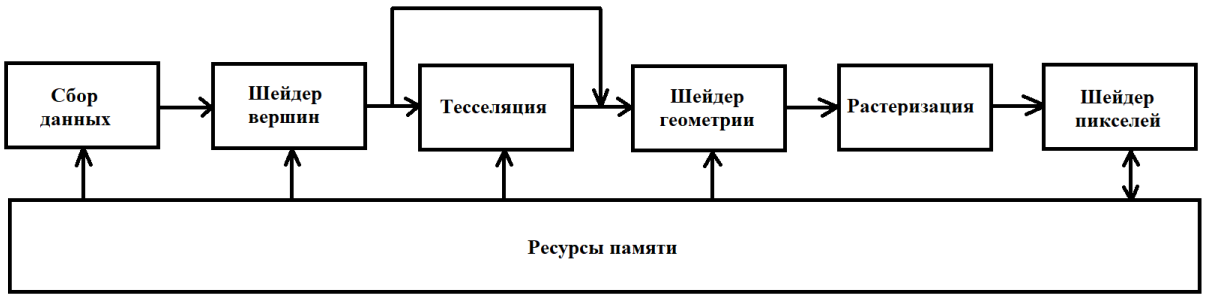


Рис. 1.14. Графический конвейер

Первая стадия графического конвейера – это *стадия сбора данных*. На этой стадии определяются треугольники или линии, которые будут выведены на экран, а также определяется семантика этих данных.

Стадия вершинного шейдера является программируемой и предоставляет возможность обработки вершин, в том числе, например, преобразование трехмерных координат вершины в координаты экранного пространства. Следует отметить, что **координаты экранного пространства** – двумерные координаты вершины на экране и еще расстояние от экрана до вершины (глубина).

Стадия тесселяции не является обязательной и дает возможность разбиения больших треугольников на маленькие.

Стадия шейдера геометрии является программируемой и позволяет организовывать новые потоки треугольников и линий.

Стадия растеризации подготавливает примитивы к выводу на экран и определяет, какие пикселы будут выведены, а какие нет.

Стадия шейдера пикселов является программируемой, получает на входе интерполированные данные для каждого пиксела (из преобразованных данных вершин) и позволяет создавать непосредственно цвет пиксела на экране.

Наиболее часто осуществляют программирование стадий вершинного или пиксельного шейдеров. Но эти стадии не являются обязательными, можно использовать шейдеры по умолчанию.

1.5. Буфер глубины

Как уже было отмечено, при выводе трехмерных объектов трехмерные координаты вершины преобразуются в двумерные координаты экрана. Однако что будет в случае, если один объект полностью или частично загородит другой? Естественно, пользователь должен наблюдать тот объект, который ближе к наблюдателю. Вернее, те пикселы объекта, которые загораживают пикселы другого объекта. Чтобы реализовать правильную отрисовку трехмерной сцены, необходимо сравнивать удаленность каждого пиксела от наблюдателя. Для этой цели и служит буфер глубины, или z-буфер. При выводе пиксела на экран помимо его цветовой компоненты, которая выводится в буфер отображения, в z-буфер заносится расстояние до пиксела от наблюдателя. Перед отображением каждого пиксела производится проверка; если значение z-буфера меньше, чем значение рисуемого пиксела, такой пиксел отбрасывается и не отрисовывается. Естественно, в начале отрисовки каждого кадра происходит очистка z-буфера. Эти сравнения при выводе пикселов производятся аппаратно.

Помимо основной функции информация из заполненного буфера глубины может использоваться для получения различных специальных эффектов, например тумана, освещения, прозрачности воды и т. п.

Пользователь может включать или выключать проверки z-буфера, что позволяет получать различные эффекты. Кроме того, z-буфер может состоять из двух буферов – буфера глубины и буфера трафарета. Применение буфера трафарета будет рассмотрено в параграфах, посвященных освещению и отображению в зеркалах.

1.6. Матричные преобразования

Один из важных аспектов трехмерной графики – это аппарат преобразований. Изначально трехмерные объекты находятся в своих системах координат. Каждый из них в общем случае требуется преобразовать, чтобы представить их вместе в мировой (общей) системе координат, или *мировом пространстве* (world space). Далее необходимо мировое пространство преобразовать в *пространство вида* (view space), или *пространство камеры*, которое зависит от позиции

и ориентации камеры. Следующее преобразование зависит от проекции: перспективной (когда дальние объекты выглядят меньше, чем ближние) или ортогональной. Такое пространство называют *пространством проекции*. И в заключительной стадии все треугольники, которые не полностью попадают в окно отображения, обрезаются, такое пространство называют *экранным пространством* (screenspace). Одним из наиболее удобных математических аппаратов для всех преобразований служат матричные преобразования.

Действительно, с помощью матрицы вида 4 × 4 всегда можно преобразовать точку с координатами *x, y, z* в точку с координатами *x', y', z'*

$$[x', y', z', 1] = [x, y, z, 1] \begin{bmatrix} M_{11} M_{12} M_{13} M_{14} \\ M_{21} M_{22} M_{23} M_{24} \\ M_{31} M_{32} M_{33} M_{34} \\ M_{41} M_{42} M_{43} M_{44} \end{bmatrix},$$

$$x' = xM_{11} + yM_{21} + zM_{31} + M_{41},$$

$$y' = xM_{12} + yM_{22} + zM_{32} + M_{42},$$

$$z' = xM_{13} + yM_{23} + zM_{33} + M_{43}.$$

Наиболее популярные мировые преобразования, или преобразования в мировом пространстве, – перемещение, поворот и масштабирование.

Перемещение точки с координатами *x, y, z* в точку с координатами *x', y', z'* определяется матрицей перемещения и выглядит следующим образом:

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

где T_x, T_y, T_z – величины смещения всех вершин объекта по каждой из осей.

Матрица масштабирования объекта также задается тремя коэффициентами, являющимися масштабированием объекта по каждой из осей,

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

где S_x, S_y, S_z – коэффициенты масштабирования по каждой из осей.

Матрица поворота (вращения) уже не так однозначно задается, как две предыдущие, поскольку требует задания оси вращения.

Матрица поворота вокруг оси X на угол θ

$$[x', y', z', 1] = [x, y, z, 1]\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Матрица поворота вокруг оси Y на угол θ

$$[x', y', z', 1] = [x, y, z, 1]\begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

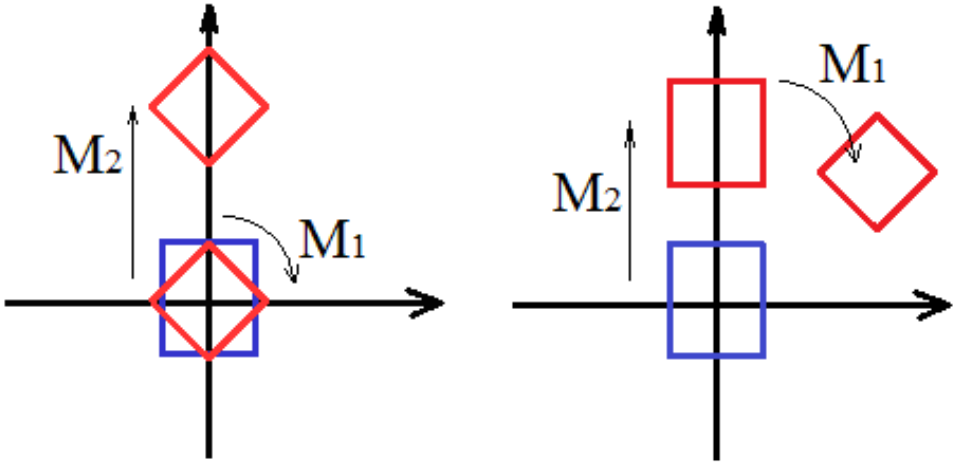
Матрица поворота вокруг оси Z на угол θ

$$[x', y', z', 1] = [x, y, z, 1]\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Преимущества применения матричного аппарата заключаются в том, что для последовательных преобразований с объектом достаточно перемножить соответствующие матрицы. Например, если требуется повернуть объект с помощью матрицы M_1, а затем переместить его с помощью матрицы M_2, то для получения общей матрицы M достаточно перемножить соответствующие матрицы $M = M_1M_2$. Следует помнить о необходимости соблюдения порядка перемножения матриц. Результаты $M = M_1M_2$ и $M = M_2M_1$ не тождественны.

Порядок трансформаций при перемножении принято записывать слева направо. Таким образом, $M = M_1M_2$ означает, что сначала произойдет трансформация M_1, а потом M_2. В нашем примере при $M = M_1M_2$ сначала произойдет вращение объекта относительно центра координат, потом его перенос в заданную точку. Если же изменить порядок перемножения матриц на $M = M_2M_1$, то сначала объект будет перенесен в какую-то точку, а потом повернут относительно центра координат. Но поскольку при переносе расстояние объекта от центра изменилось (например в 100 раз), то и объект будет повернут с радиусом в 100 раз большим и окажется совсем в другом месте, что проиллюстрировано на рис. 1.15. На рис. 1.15, *а* сначала происходит поворот синего объекта на 45° (M_1), а потом его перенос по оси Y (M_2), т. е.

общее преобразование $M = M_1M_2$. А на рис. 1.15, *б*, наоборот, сначала объект переносится по оси Y (M_2), а потом поворачивается на 45° M_1, т. е. $M = M_2M_1$. Как видим, результаты финального преобразования различны.



а) *б)*

Рис. 1.15. Порядок трансформации объекта в мировом пространстве

Надо заметить, что для трансформации вершины объекта (точки пространства) неоходимо перемножить вектор вершины (x_v, y_v, z_v, 1) и матрицу преобразования, а при трансформации нормали – вектор нормали и матрицу преобразования (x_n, y_n, z_n, 0). Разница в том, что нормали не меняются при перемещении, поэтому четвертая компонента при перемножении нулевая. Второй вариант трансформации нормали – умножение вектора нормали (x_n, y_n, z_n) на матрицу трансформации, но размерностью 3 × 3, т. е. исключая последний столбец и последнюю строку. Кроме этого нормали обычно являются нормализованными векторами с длиной, равной единице, поэтому после трансформации их в большинстве случаев необходимо нормализовать.

Еще одна матрица, которая потребуется для построения трехмерных сцен, – *матрица вида*, или *матрица камеры*. Эта матрица переносит всю геометрию в пространство вида. В пространстве вида камера находится в начале координат и смотрит вдоль положительной оси Z (необходимо отметить, что в DirectX в отличие от OpenGL, используется левосторонняя система координат).

Один из общих подходов к вычислению матрицы вида состоит в последовательном вращении камеры вокруг каждой оси и переносе камеры в заданную точку. Последовательнось описывается уравнением $M_v = TR_xR_yR_z$, где M_v – искомая матрица вида; $R_xR_yR_z$ – матрицы поворотов вокруг осей X, Y, Z; T – матрица переноса в заданную точку. Причем следует понимать, что если необходимо переместить камеру в точку с координатами (10; 0; 5), то матрица трансформации должна строиться для вектора переноса (–10; 0; –5). Таким образом, необходимо сначала нужную точку (ту, где будет камера) перенести в начало координат, а потом уже ее вращать. Другими словами, это не камера путешествует по миру, а мир подстраивается под условия камеры. Поэтому направления вращения тоже строятся из отрицательных углов поворота. Если камеру желательно повернуть вокруг оси Y на 30°, это значит, что мир должен повернуться относительно камеры (т. е. центра координат) на –30°.

При построении матрицы проекции необходимо исходить из того, какая проекция необходима: ортогональная или перспективная. В трехмерной графике при визуализации трехмерных сцен чаще всего используется перспективная проекция, хотя встречаются задачи, в которых необходима и матрица ортогональной проекции. Матрица перспективной проекции выглядит следующим образом:

$$\begin{bmatrix} w & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & \theta & 1 \\ 0 & 0 & -\theta Z_n & 0 \end{bmatrix}, \quad \begin{matrix} w = ctn(f_{\text{w}} / 2) \\ h = ctn(f_{\text{h}} / 2) \\ \theta = \dfrac{Z_f}{Z_f - Z_n} \end{matrix},$$

где f_w и f_h – углы по ширине и высоте обзора; Z_f – расстояние до дальней плоскости отсечения (максимальное расстояние, до которого камера производит отображение объектов); Z_n – расстояние до ближней плоскости отсечения (минимальное расстояние, с которого камера начинает отображение объектов). Зная размеры экрана или окна отображения, вычислить значения w и h можно так:

$$w = \frac{2Z_n}{V_w}, h = \frac{2Z_n}{V_h},$$

где V_w и V_h – размеры окна вывода по ширине и высоте.

Матрица перспективной проекции предполагает, что значения x и y применительно к окну отображения меняются в диапазоне от –1 до 1, а значение z меняется от 0 до 1 (0 соответствует Z_n, а 1 соответствует Z_f). Когда задаем Z_n, необходимо исходить из того, что чем выше это значение, тем лучше. Связано это с тем, что после всех преобразований значения z будут нелинейны и вблизи Z_n, т. е. вблизи нуля, слабо отличаться друг от друга, что при низкой разрядности буфера глубины может породить негативные эффекты.

Итак, прежде чем визуализировать трехмерный объект средствами DirectX, необходимо вычислить матрицу трансформации, которая равна

$$M = M_w M_v M_p,$$

где M_w – матрица мирового преобразования; M_v – матрица вида; M_p – матрица проекции. Применение к объекту (ко всем его вершинам) итоговой матрицы M переводит объект в экранное пространство.

В математической библиотеке directxmath присутствуют все необходимые функции для работы с матрицами: вычисление матриц вида, матрицы перспективной и ортогональной проекций, матриц трансформаций и перемножений. Также присутствуют функции вычисления инвертированной, транспонированной, единичной матриц и матрицы отражения.

Контрольные вопросы

1. Какими примитивами оперирует библиотека DirectX?
2. Что такое вершина?
3. Какими атрибутами может обладать вершина?
4. Что такое нормаль?
5. В каком случае вершины треугольника будут иметь разные нормали?
6. Для чего используют нормаль?
7. Что такое текстурные координаты?
8. Что такое mip уровни текстуры?
9. Зачем осуществляют фильтрацию при наложении текстур?

10. Каковы основные стадии графического конвейера?

11. Какие стадии графического конвейера являются программируемыми?

12. На каком языке осуществляется программирование некоторых этапов графического конвейера?

13. Что такое буфер глубины?

14. Зачем нужен буфер глубины?

15. Что такое мировое пространство?

16. Что такое пространство вида?

17. Что такое экранное пространство?

18. Перечислите основные типы мировых преобразований.

19. Что такое перспективная проекция?

20. Что такое ортогональная проекция?

21. Что такое ближняя и дальняя плоскости отсечения?

22. Вычислите матрицу поворота вокруг оси X на 30°.

23. Вычислите матрицу поворота объекта вокруг оси Y на 60° и сдвиг его вдоль этой оси на единицу.

24. Какая функция с матрицами отдельных трансформаций объекта вычисляет общую матрицу трансформации?

25. В каком порядке следует перемножать матрицы трансформации?

Глава 2. ОСНОВЫ ПРОГРАММИРОВАНИЯ СРЕДСТВАМИ DIRECTX 11

2.1. Настройка Microsoft Visual Studio

Программирование трехмерной графики с применением библиотеки DirectX 11 рассчитано на использование языка программирования C++. Удобнее всего программировать в среде Microsoft Visual Studio. В настоящем учебном пособии приведены примеры с использованием Visual Studio 19, впрочем, подойдут и более поздние версии.

Сначала необходимо скачать и установить DirectX SDK 11 с сайта компании Microsoft. Также потребуются библиотеки работы с текстурами, которые можно скачать по ссылкам: https://www.nuget.org/packages/directxtk_desktop_win10/2022.7.30.1, https://www.nuget.org/packages/directxtex_desktop_win10/2022.7.30.1.

Далее в свойствах проекта следует установить пути к директориям необходимых библиотек, как показано на рис. 2.1.

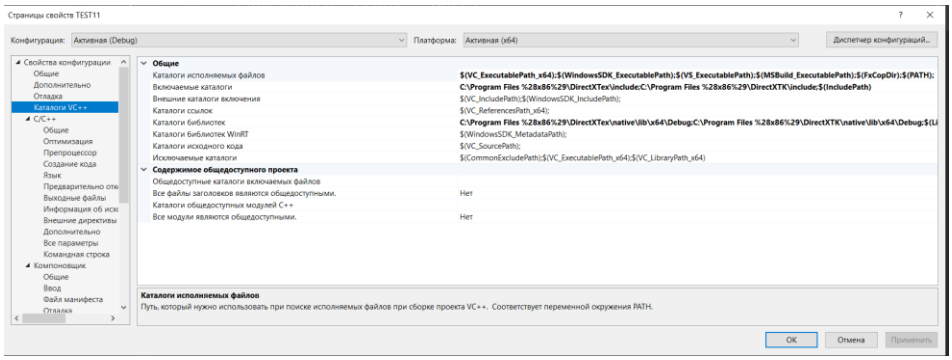


Рис. 2.1. Настройка свойств конфигурации проекта

Рекомендуется также установить библиотеку FBX SDK, которая потребуется при программировании анимации трехмерных объектов. Подключение библиотеки FBX SDK к проекту показано на рис. 2.2.

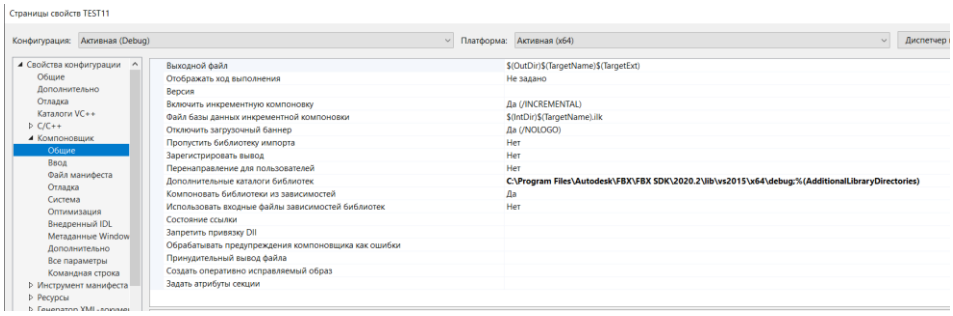


Рис. 2.2. Дополнительные каталоги библиотек

Во вкладке компоновщика, как показано на рис. 2.3, необходимо установить дополнительные зависимости, что впоследствии избавит от проблем с компиляцией.

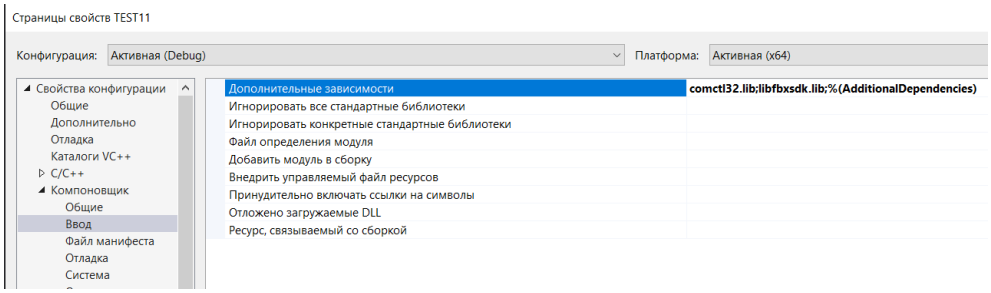


Рис. 2.3. Дополнительные зависимости

Целесообразно подключить к создаваемому проекту заголовочные файлы математических библиотек: directxmath.h, DirectXCollision.h. Эти библиотеки поддерживают быстрые 64-разрядные инструкции процессора.

2.2. Инициализация объекта DirectX

В зависимости от задачи разрабатываемое приложение должно содержать одно или несколько окон, которые будут использованы библиотекой DirectX 11 для отображения графики. В принципе, при инициализации объекта DirectX потребуется только дескриптор (HWND) окна, в которое и будет направлен поток вывода графических данных.

Создадим собственную функцию инициализации, которая вернет значение S_OK, если все успешно, и E_FAIL, если инициализация завершилась ошибкой

HRESULT Initialize (int winwidth, int winheight, bool vsync, HWND hwnd, bool fullscreen),

где winwidth, iwinheight – ширина и высота окна; vsync – наличие вертикальной синхронизации; hwnd – дескриптор окна; fullscreen – наличие полноэкранного режима.

Разберем шаги инициализации объекта DirectX.

1. Вызовем функцию, создающую фабрику интерфейсов DirectX,

CreateDXGIFactory(__uuidof(IDXGIFactory), (void**)&factory),

где factory – указатель на фабрику интерфейсов.

2. Получим интерфейс основного графического адаптера (видеокарты)

factory->EnumAdapters(0, &adapter),

где adapter – указатель на интефейс графического адаптера.

3. Получим интерфейс основного устройства вывода (монитора)

adapter->EnumOutputs(0, &adapterOutput),

где adapterOutput – указатель на интерфейс основного устройства вывода.

4. Получим число режимов отображения в формате RGBA

adapterOutput->GetDisplayModeList
(DXGI_FORMAT_R8G8B8A8_UNORM,
DXGI_ENUM_MODES_INTERLACED, &numModes, NULL),

где numModes – число режимов отображения.

Получим список всех режимов отображения в формате ARGB

adapterOutput->GetDisplayModeList
(DXGI_FORMAT_R8G8B8A8_UNORM,
DXGI_ENUM_MODES_INTERLACED, &numModes, display-ModeList),

где displayModeList – массив всех режимов, поддерживаемых адаптером и устройством отображения.

5. Найдем режим, совпадающий с разрешением экрана, и определим параметры обновления экрана numerator и denominator

```
UINT screenWidth = GetSystemMetrics(SM_CXSCREEN);
UINT screenHeight = GetSystemMetrics(SM_CYSCREEN);
unsigned int i, numerator, denominator;
for (i = 0; i<numModes; i++)
{
      if (displayModeList[i].Width == (unsigned int)screenWidth)
      {
       if (displayModeList[i].Height == (unsigned int)screenHeight)
       {
            numerator = displayModeList[i].RefreshRate.Numerator;
            denominator = displayModeList[i].RefreshRate.Denominator;
       }
      }
}
```

6. Прочитаем описание видеокарты и размер ее памяти в Мб

adapter->GetDesc(&adapterDesc),

где adapterDesc – структура описывающая видеокарту.

videoCardMemory = (int)(adapterDesc.DedicatedVideoMemory / 1024 / 1024);

int error=wcscpy_s(videoCardDescription, adapterDesc.Description);

В массиве символов videoCardDescription будет лежать название видеокарты.

7. Удалим ненужные более интерфейсы

adapterOutput->Release();

adapter->Release();

factory->Release();

8. Создадим устройство (device), контекст устройства (device-context) и цепочку переключения буферов (swapchain), заполнив необходимую структуру. Надо заметить, что понадобится минимум два буфера отображения. Один буфер передний – это то, что видит пользователь; а другой – задний. Пока пользователь видит картинку на переднем буфере, программа готовит изображение в заднем буфере. Как только изображение будет готово, буферы поменяются местами. Передний станет задним, там будет готовиться следующий кадр, а задний станет передним. Цепочка переключения (swapchain) как раз и устанавливает правила переключения переднего и заднего буферов.

Например, если включена вертикальная синхронизация, то момент переключения буферов будет совпадать с моментом начала вертикальной развертки кадра и число кадров в секунду совпадет с частотой обновления экрана пользователя.

D3D11CreateDeviceAndSwapChain(NULL, D3D_DRIVER_TYPE_HARDWARE, NULL, 0, &featureLevel, 1, D3D11_SDK_VERSION, &swapChainDesc, &this->swapChain, &this->device, NULL, &this->deviceContext)

9. Создадим поверхность отображения

swapChain->GetBuffer(0, __uuidof(ID3D11Texture2D), (LPVOID*)&pointer) device->CreateRenderTargetView(pointer, NULL, & targets),

где targets – указатель на поверхность отображения.

10. Создадим буфер глубины так, чтобы его можно было использовать еще и как текстуру

device->CreateTexture2D(&depthBufferDesc, NULL, &pointer),

где depthBufferDesc – структура, описывающая параметры буфера глубины.

device->CreateShaderResourceView(pointer, &descSRV,

&depthStencilTexture),

где depthStencilTexture – указатель на текстуру с данными глубины; descSRV – структра описания этого ресурса.

device->CreateDepthStencilView(pointer, &depthStencilViewDesc, & depth_stencil),

где depth_stencil – указатель на поверхность буфера глубины, depthStencilViewDesc-структура описания этого ресурса.

11. Установим вьюпорт (окно вывода DirectX)

deviceContext->RSSetViewports(1, &viewport),

где viewport – структура с параметрами окна вывода.

12. Установим цель рендеринга и буфер глубины для конвейера рендеринга

deviceContext->OMSetRenderTargets(1, &target, depth_stencil);

После инициализации DirectX основной цикл рендеринга будет выглядеть следующим образом.

1. Очистка поверхности рендеринга и буфера глубины

deviceContext->ClearRenderTargetView(target, (FLOAT*)color),

где color – массив из четырех значений float, задающих цвет в формате RGBA.

deviceContext->ClearDepthStencilView(depth_stencil,

D3D11_CLEAR_DEPTH | D3D11_CLEAR_STENCIL, 1.0f, 0);

2. Вывод трехмерной сцены.

3. Представление созданной трехмерной сцены на экране пользователя. В зависимости от наличия вертикальной синхронизации либо ждем начала обновления кадра в соответствии с частотой кадровой развертки, либо отображаем немедленно

if (vsync_enabled) swapChain->Present(1; 0);

else swapChain->Present(0; 0);

В прил. 1 приведены функция инициализации и функция основного цикла отрисовки на основе класса D3D. Пример инициализации объекта DirectX и окрашивания рабочего окна в различные цвета

изображен на рис. 2.4. Исходные коды для примера можно скачать по ссылке: https://github.com/samoyow1973/Example1.

Рис. 2.4. Рабочее окно примера инициализации приложения

2.3. Вершинные и индексные буферы

Как уже было показано выше, трехмерные объекты состоят из вершин. Для отрисовки трехмерного объекта необходимо создать буфер вершин, в котором будут храниться все вершины со всеми атрибутами. Этот буфер будет источником информации для конвейера рендеринга. Поскольку трехмерные объекты состоят из треугольников, конвейер рендеринга будет брать по три вершины для отображения очередного треугольника, из которого состоит трехмерный объект.

Однако DirectX позволяет делать выборки не по три вершины для каждого треугольника, но и другим способом. Например, четырехугольник, показанный на рис. 2.5, состоит из двух треугольников с вершинами *ABC* и *BCD*. DirectX позволяет отображать такой объект двумя способами, передавая графическому конвейеру по три вершины (в нашем примере *ABC* и *BCD*), или, если следующий треугольник содержит две вершины предыдущего треугольника, передавая недостающую вершину. Достигается это за счет кэша вершин, который содержат все современные графические процессоры. Для нашего примера будут переданы вершины *A*, *B*, *C* для первого треугольника и вершина *D* – для второго. Таким образом, будут переданы вершины *A*, *B*, *C*, *D*.

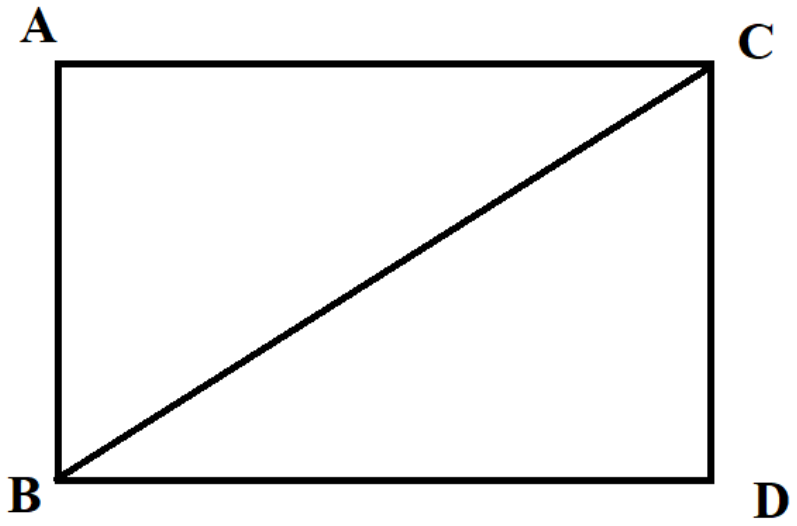


Рис. 2.5. Пример объекта, разбитого на треугольники

Во втором способе в графический конвейер передается меньше вершин, особенно если трехмерный объект состоит из большого числа треугольников, как показано на рис. 2.6. В этом случае будут переданы вершины *A*, *B*, *C*, *D*, *E*, *F* и нарисованы треугольники *ABC*, *BCD*, *CDE*, *DEF*. Таким образом, для вывода четырех треугольников потребовалось шесть вершин. Следует понимать, что чем меньше информации передается по шине данных графического конвейера, тем выше быстродействие и тем более сложные трехмерные графические сцены удастся выводить в реальном времени.

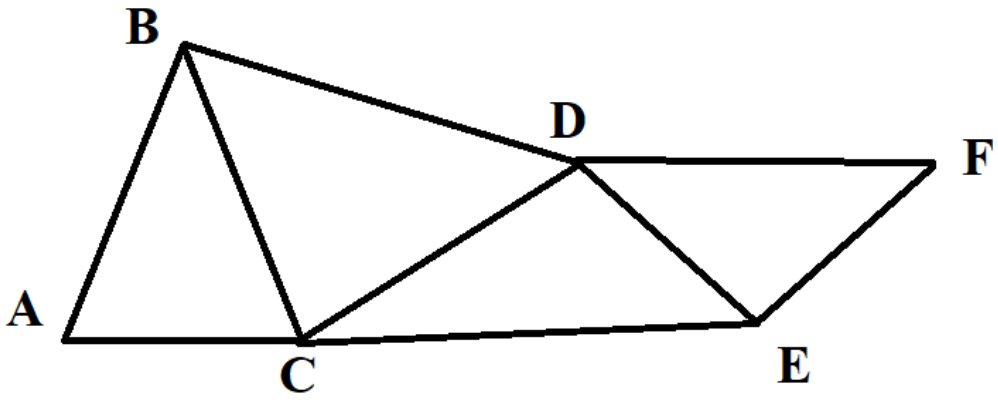


Рис. 2.6. Объект, содержащий шесть вершин и четыре треугольника

Тем не менее не всегда удается представить объект с помощью списка треугольников, где каждый следующий треугольник содержит вершину предыдущего. Кроме того, вершина может содержать большое количество атрибутов, что значительно снизит скорость передачи информации. Поэтому есть еще один вариант вывода треугольников графического объекта. В этом варианте передаются не сами вершины со всеми атрибутами, а только их индексы (номера) в буфере вершин. В этом случае помимо буфера вершин необходимо создать в памяти буфер индексов вершин, из которых состоит треугольник. Именно из индексного буфера и будет осуществляться передача информации о том, какие треугольники выводить. При индексном выводе также возможны два варианта: передавать для каждого треугольника по три индекса вершин или, если следующий треугольник содержит две вершины предыдущего, передавать только недостающий индекс.

Рассмотрим создание вершинного и индексного буферов для простейшего примера при выводе одного треугольника с вершинами разных цветов.

1. Создадим структуру с атрибутами вершин

```
struct VERTEX
{
   XMFLOAT2 pos; // позиция вершины на экране
   XMFLOAT4 color; // цвет вершины в формате RGBA
};
```

2. Создадим массив вершин и индексов и заполним их

```
VERTEX Triangle[3];
Triangle[0].pos.x = 0.0f;
Triangle[0].pos.y = 1.0f;
Triangle[0].color = XMFLOAT4(1.0f, 0.0f, 0.0f, 1.0f);
Triangle[1].pos.x = 0.5f;
Triangle[1].pos.y = 0.0f;
Triangle[1].color = XMFLOAT4(0.0f, 1.0f, 0.0f, 1.0f);
Triangle[2].pos.x = 1.0f;
Triangle[2].pos.y = 1.0f;
Triangle[2].color = XMFLOAT4(0.0f, 0.0f, 1.0f, 1.0f);
unsigned long Indices[4];
Indices[0] = 0;
Indices[1] = 1;
Indices[2] = 3;
```

3. Заполним структуры, описывающие параметры и атрибуты создаваемых буферов вершин и индексов,

D3D11_BUFFER_DESC vertexBufferDesc, indexBufferDesc;

D3D11_SUBRESOURCE_DATA vertexData, indexData;

Заполним структуру, описывающую буфер вершин, причем сделаем последний динамическим с возможностью записи в него из CPU, чтобы в дальнейшем была возможность менять координаты вершин

vertexBufferDesc.Usage = D3D11_USAGE_DYNAMIC;

vertexBufferDesc.ByteWidth = sizeof(VERTEX) * 3;

vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;

vertexBufferDesc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;

vertexBufferDesc.MiscFlags = 0;

vertexBufferDesc.StructureByteStride = 0;

Заполним субструктуру, описывающую память, где хранятся вершины треугольника.

vertexData.pSysMem = Triangle;

vertexData.SysMemPitch = 0;

vertexData.SysMemSlicePitch = 0;

Заполним структуру, описывающую буфер индексов, причем сделаем его статическим, поскольку нет необходимости менять порядок вывода вершин

indexBufferDesc.Usage = D3D11_USAGE_DEFAULT;

indexBufferDesc.ByteWidth = sizeof(unsigned long) * 3;

indexBufferDesc.BindFlags = D3D11_BIND_INDEX_BUFFER;

indexBufferDesc.CPUAccessFlags = 0;

indexBufferDesc.MiscFlags = 0;

indexBufferDesc.StructureByteStride = 0;

Заполним субструктуру, описывающую память, где хранятся индексы вершин треугольника,

indexData.pSysMem = Indices;

indexData.SysMemPitch = 0;

indexData.SysMemSlicePitch = 0;

4. Создадим буфер вершин

device->CreateBuffer(&vertexBufferDesc, &vertexData, &VBuffer);

5. Создадим буфер индексов

device->CreateBuffer(&indexBufferDesc, &indexData, &IBuffer);

Теперь в переменных VBuffer и IBuffer лежат указатели на буферы вершин и индексов. Чтобы отправить трехмерный объект графическому конвейеру, необходимо указать, в каких буферах лежат вершины и их индексы

```
unsigned int stride, offset;
stride = sizeof(VERTEX);
offset = 0;
deviceContext->IASetVertexBuffers(0, 1, &VBuffer, &stride, &offset);
deviceContext->IASetIndexBuffer(IBuffer,
DXGI_FORMAT_R32_UINT, 0);
```

Переменная offset содержит отступ от начала вершинного буфера и обычно равна нулю. Переменная stride содержит объем памяти, занимаемой одной вершиной, в байтах. Далее необходимо установить тип вывода треугольников списком

D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST

или в случае, когда каждый следующий треугольник содержит две последние вершины предыдущего,

D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP

Этот параметр можно указать функцией

```
deviceContext->IASetPrimitiveTopology(
D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP)
```

Вывод индексированных примитивов осуществляется функцией

```
deviceContext->DrawIndexed(IndexCount, StartIndexLocation,
BaseVertexLocation),
```

где IndexCount – количество индексов, которое будет посчитано, обычно IndexCount = к + 2, где к – число выводимых треугольников; StartIndexLocation – отступ от начала буфера индексов, обычно 0; BaseVertexLocation – отступ от начала вершинного буфера, обычно 0.

Вывод неиндексированных треугольников осуществляется функцией

```
deviceContext->Draw(VertexCount,StartIndexLocation),
```

где VertexCount – число вершин, которое будет выведено, обычно VertexCount=k*3; StartIndexLocation – отступ от начала буфера вершин.

Различные параметры отступов необходимы для случая, когда в одном буфере находятся вершины или индексы различных трехмерных объектов и вывод этих объектов будет происходить по отдельности. Следует отметить, что производительность будет тем выше, чем

меньшее число функций Draw или DrawIndexed использовано. При выводе объектов с одинаковой геометрией (т. е. одинаковыми буферами вершин и индексов) можно использовать встроенную в DirectX оптимизацию с целью минимизации вызовов функций Draw или DrawIndexed, о которой речь пойдет в конце пособия.

2.4. Управление состояниями конвейера рендеринга

Некоторые стадии конвейера рендеринга напрямую программировать нельзя, но можно менять их алгоритмы работы за счет изменения некоторых параметров. К таким стадиям относятся стадия растеризации, стадия управления буфером глубины и трафарета, стадия текстурирования и стадия смешивания. Сформированных параметров для каждой стадии может быть несколько (в зависимости от количества различных графических техник и приемов), и устанавливаются они перед вызовом функций Draw() или DrawIndexed(). Рассмотрим, как устанавливать и изменять параметры каждой стадии.

Параметры стадии растеризации устанавливаются с помощью заполнения структуры D3D11_RASTERIZER_DESC. Структура содержит следующие поля:

AntialiasedLineEnable – сглаживание линий;

CullMode – указывает, какие треугольники рисовать: те, которые обращены лицевой стороной или обратной, или можно рисовать любые;

DepthBias – значение глубины, добавляемое к пикселу (для рисования, например, граффити на стенах);

DepthBiasClamp – максимальная глубина, добавляемая к пикселу;

DepthClipEnable – включить обрезание в зависимости от расстояния;

FillMode – рисовать закрашенные треугольники или только их ребра;

FrontCounterClockwise – направление вершин, определяющее лицевую сторону треугольника;

MultisampleEnable – наличие мультисэмплирования (сглаживания);

ScissorEnable – включить отсечение пикселей за пределами прямоугольника;

SlopeScaledDepthBias – добавление глубины в зависимости от угла.

Заполненная структура позволяет создать состояние растеризации с помощью функции

device->CreateRasterizerState(&desc, &Rasterstate),

где desc – заполненная структура, Rasterstate – указатель на состояние стадии растеризации с типом ID3D11RasterizerState*.

Установить состояние растеризации можно функцией

deviceContext->RSSetState(Rasterstate).

Пример заполнения структуры для определения параметров растеризации выглядит следующим образом:

```
D3D11_RASTERIZER_DESC descr;
descr.AntialiasedLineEnable = FALSE;
descr.CullMode = D3D11_CULL_NONE;
descr.DepthBias = 0;
descr.DepthBiasClamp = 0.0f;
descr.DepthClipEnable = 1;
descr.FillMode = D3D11_FILL_SOLID;
descr.FrontCounterClockwise = 0;
descr.MultisampleEnable = 0;
descr.ScissorEnable = 0;
descr.SlopeScaledDepthBias = 0.0f;
```

Параметры стадии управления буфером глубины и трафарета устанавливаются с помощью заполнения структуры D3D11_DEPTH_STENCIL_DESC, которая содержит следующие поля:

DepthEnable – включить/выключить буфер глубины;

DepthWriteMask – включить/выключить запись в буфер глубины;

DepthFunc – функция сравнения значения глубины пикселя и значения из буфера;

StencilEnable – включить/выключить буфер трафарета;

StencilReadMask – маска для чтения из буфера трафарета;

StencilWriteMask – маска для записи в буфер трафарета.

Следующие поля относятся к выводимым треугольникам, которые находятся лицевой стороной:

FrontFace.StencilFailOp – тип изменения значения в буфере трафарета, если тест трафарета не пройден;

FrontFace.StencilDepthFailOp – тип изменения значения в буфере трафарета, если тест буфера глубины не пройден, а тест трафарета пройден;

FrontFace.StencilPassOp – тип изменения значения в буфере трафарета, если тест трафарета и глубины пройден;

FrontFace.StencilFunc – функция теста трафарета.

Следующие поля относятся к отрисовываемым треугольникам, которые находятся обратной стороной к камере:

BackFace.StencilFailOp – тип изменения значения в буфере трафарета, если тест трафарета не пройден;

BackFace.StencilDepthFailOp – тип изменения значения в буфере трафарета, если тест буфера глубины не пройден;

BackFace.StencilPassOp – тип изменения значения в буфере трафарета, если тест трафарета пройден;

BackFace.StencilFunc – функция теста трафарета;

Создать состояния настроек буфера глубины и трафарета можно функцией

device->CreateDepthStencilState(&desc, &Depthstencilstate),

где desc – заполненная структура; Depthstencilstate – указатель на состояние настроек буфера глубины и трафарета с типом ID3D11DepthStencilState*.

Установить состояние растеризации можно функцией

deviceContext->OMSetDepthStencilState(Depthstencilstate, value),

где value – значение для теста трафарета.

Подробно управление буфером трафарета и буфером глубины будет рассмотрено в следующих параграфах. Пример заполнения структуры для определения параметров растеризации выглядит следующим образом:

```
D3D11_DEPTH_STENCIL_DESC descz;
descz.DepthEnable = 0;
descz.DepthWriteMask = D3D11_DEPTH_WRITE_MASK_ALL;
descz.DepthFunc = D3D11_COMPARISON_LESS;
descz.StencilEnable = 0;
descz.StencilReadMask = 255;
descz.StencilWriteMask = 255;
descz.FrontFace.StencilFailOp = D3D11_STENCIL_OP_KEEP;
descz.FrontFace.StencilDepthFailOp = D3D11_STENCIL_OP_INCR;
```

descz.FrontFace.StencilPassOp = D3D11_STENCIL_OP_KEEP;
descz.FrontFace.StencilFunc = D3D11_COMPARISON_ALWAYS;
descz.BackFace.StencilFailOp = D3D11_STENCIL_OP_KEEP;
descz.BackFace.StencilDepthFailOp = D3D11_STENCIL_OP_DECR;
descz.BackFace.StencilPassOp = D3D11_STENCIL_OP_KEEP;
descz.BackFace.StencilFunc = D3D11_COMPARISON_ALWAYS;

Параметры стадии управления текстурированием устанавливаются с помощью заполнения структуры D3D11_SAMPLER_DESC, которая содержит следующие поля:

Filter – метод фильтрации текстур;

AddressU – метод определения текстурной координаты в случае выхода за диапазон [0; 1];

AddressV – метод определения текстурной координаты в случае выхода за диапазон [0; 1];

AddressW – метод определения текстурной координаты в случае выхода за диапазон [0; 1];

MipLODBias – смещение для вычисления mipmap уровня;

MaxAnisotropy – уровень анизотропной фильтрации;

ComparisonFunc – функция сравнения данных текстуры с внешними данными;

BorderColor – массив из четырех значений цвета в формате RGBA для рамки при текстурных координатах вне диапазона [0; 1];

MinLOD – нижнее значение уровней mipmap;

MaxLOD – верхнее значение уровней mipmap.

Создать состояния настроек для управления текстурированием можно функцией

device->CreateSamplerState(&desc, &Samplestate),

где desc – заполненная структура; Samplestate – указатель на состояние текстурирования с типом ID3D11SamplerState*.

Установить состояние текстурирования можно функцией

deviceContext->PSSetSamplers(slot, count, &SampleState),

где slot – номер текстурного слота; count – число состояний текстурирования; &SampleState – массив настроек текстурирования.

Подробно управление текстурированием будет рассмотрено в следующих параграфах.

Параметры стадии смешивания устанавливаются с помощью заполнения структуры D3D11_BLEND_DESC, которая содержит следующие поля:

AlphaToCoverageEnable – определяет, будет ли использовано смешивание при мультисэмплировании;

IndependentBlendEnable – определяет, будет ли применено смешивание для для целей отрисовки больше 0;

RenderTarget[j].BlendEnable – включить/выключить смешивание;

RenderTarget[j].SrcBlend – первый аргумент RGB;

RenderTarget[j].DestBlend – второй аргумент RGB;

RenderTarget[j].BlendOp – определяет функцию комбинирования RGB аргументов;

RenderTarget[j].SrcBlendAlpha – первый альфа-аргмент;

RenderTarget[j].DestBlendAlpha – второй альфа-аргумент;

RenderTarget[j].BlendOpAlpha – определяет функцию смешивания альфа аргументов;

RenderTarget[j].RenderTargetWriteMask – маска для цветов в формате RGBA.

Создать состояния настроек смешивания можно функцией

device->CreateBlendState(&desc, &Blendstate),

где desc – заполненная структура; Blendstate – указатель на состояние настроек смешивания с типом ID3D11BlendState*.

Установить состояние настроек смешивания можно функцией

deviceContext->OMSetBlendState(Blendstate, blendFactor, mask),

где blendFactor – фактор смешивания для каждого из четырех компонентов RGBA; mask – маска при мультисэмплировании.

Подробно управление смешиванием будет рассмотрено в следующих параграфах и главах.

2.5. Вершинные шейдеры

Вершинные шейдеры – это программируемый этап конвейера рендеринга, где производится обработка вершин. В общем случае вершины трехмерного объекта необходимо преобразовать в соответствии с матрицей проекции, матрицей камеры и матрицей трансформации объекта. Кроме того, в вершинных шейдерах можно рассчитывать освещение вершин, видоизменять их положение в пространстве и

производить множество других операций. Тема шейдеров (как вершинных, так и пиксельных) настолько объемная, что выходит за рамки настоящего учебного пособия. В этом параграфе разберем пример создания простейшего шейдера, его загрузки и компиляции.

При программировании с помощью библиотеки DirectX целесообразно писать шейдеры на языке HLSL [28; 31]. Этот язык очень похож на язык программирования C++ и не вызывает особых трудностей. Вершинный шейдер на входе получает вершину, а именно список (возможно, частичный) ее атрибутов из вершинного буфера. Например, вершинный шейдер для треугольника из предыдущего параметра будет получать позицию вершины в виде двумерного вектора (координаты вершины на экране) и цвет вершины в виде четырехмерного вектора в формате RGBA

```
struct INPUT
{
 float2 position : POSITION;
 float4 color : COLOR0;
};
```

Выходные данные, которые будут интерполироваться между вершинами и поступать на вход пиксельного шейдера, тоже будут содержать позицию на экране и цвет пиксела

```
struct OUTPUT
{
 float4 position : SV_POSITION;
 float4 color : COLOR0;
};
```

Система координат x и y представляет собой декартову систему координат с центром в середине окна (0; 0), координатами левого верхнего угла окна (–1; 1) и координатами (1; –1) в правом нижнем углу окна.

Необходимо отметить, что позиция на экране содержит уже четырехкомпонентный вектор. Это связано с тем, что кроме координат экрана необходимо передавать пиксельному шейдеру расстояние от наблюдателя до пиксела. В нашем примере в этом нет надобности, поэтому две дополнительные координаты z и w будут приравнены к нулю и единице соответственно, что эквивалентно плоскости экрана монитора.

Основная функция шейдера будет выглядеть следующим образом:

```
OUTPUT main (INPUT input)
{
 OUTPUT output;
 output.position.xy = input.position;
 output.position.zw = float2(0.0, 1.0);
 output.color = input.color;
 return output;
}
```

В нашем примере вершинный шейдер просто копирует данные. В принципе, можно было и не применять его вообще, но на этом примере рассмотрим, как создавать, компилировать и загружать вершинные шейдеры. Добавим в проект файл с типом «вершинный шейдер» и скопируем в него вышеприведенный текст программы на языке HLSL. Откроем свойства файла, как показано на рис. 2.7 и в поле точки входа установим функцию main, а версию шейдера выберем 5.0.

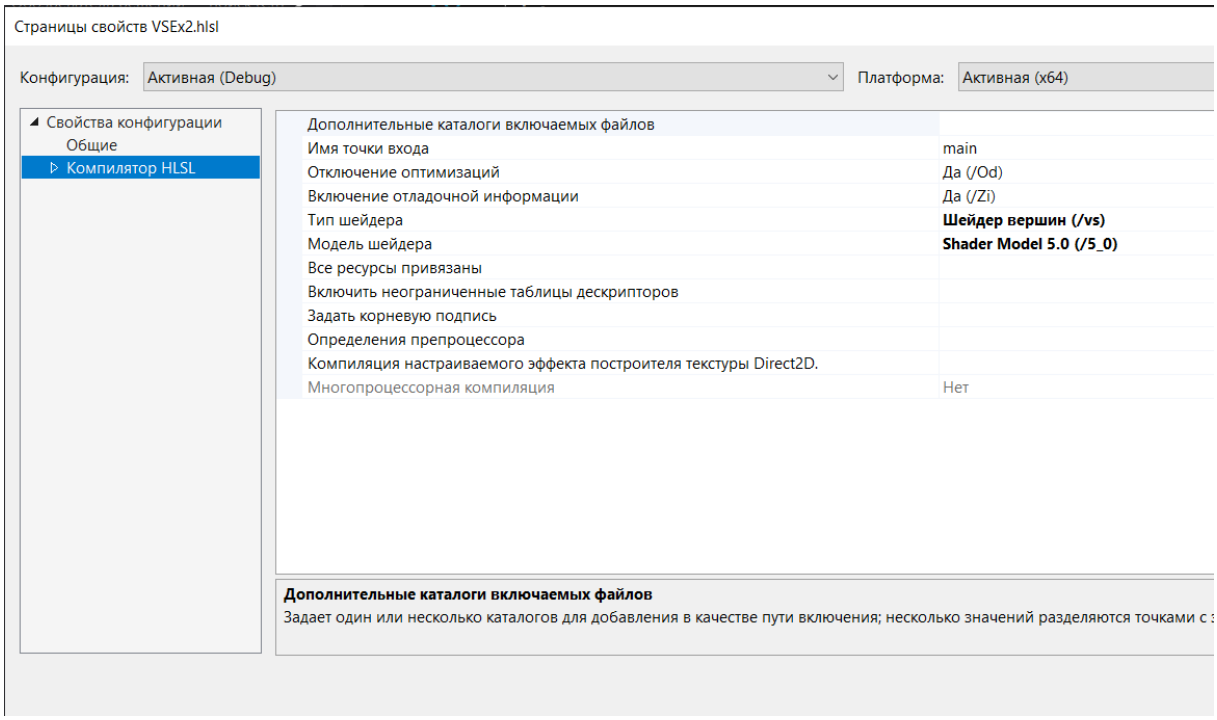


Рис. 2.7. Свойства вершинного шейдера

Есть два варианта загрузки шейдера. Первый вариант – загрузка нескомпилированного шейдера и последующая его компиляция; второй вариант – загрузка уже скомпилированного шейдера.

В первом варианте воспользуемся функцией

D3DCompileFromFile(name, NULL, NULL, "main", "vs_5_0", D3D10_SHADER_ENABLE_STRICTNESS, 0,&vertexShaderBuffer, &errorMessage),

где name – это путь и имя файла с расширением .hlsl, а vertexShader-Buffer и errorMessage – указатели на область памяти с типом ID3D10Blob*, где будут находиться шейдер и сообщение об ошибке, если компиляция не удастся.

Во втором варианте загрузим уже скомпилированный шейдер

D3DReadFileToBlob(name, &vertexShaderBuffer),

где name – путь и имя файла с расширением .cso.

После компиляции или загрузки воспользуемся функцией непосредственно создания вершинного шейдера

device->CreateVertexShader(vertexShaderBuffer->GetBufferPointer(),
vertexShaderBuffer->GetBufferSize(), NULL,vertexShader),

где vertexShader – указатель типа ID3D11VertexShader* уже непосредственно на сам шейдер, которым и будем пользоваться в дальнейшем.

Теперь необходимо рассмотреть вопрос, как данные из буфера вершин попадают в вершинный шейдер. Дело в том, что конкретному шейдеру не обязательно необходимы все атрибуты вершины. Например, если цель вывода трехмерных объектов состоит только в том, чтобы заполнить буфер глубины (для последующих эффектов), то вершинному шейдеру необходимы только позиции вершин в пространстве, а атрибуты текстурных координат, нормалей и прочее не нужны. Поэтому каждый вершинный шейдер должен иметь представление о том, какие атрибуты вершины брать из вершинного буфера. Для этого необходимо заполнить массив структур D3D11_INPUT_ELEMENT_DESC, количество элементов которого равно количеству необходимых атрибутов вершины для вершинного шейдера, например,

```
D3D11_INPUT_ELEMENT_DESC* desc=NULL;
UINT numElements;
numElements=2;
desc = new D3D11_INPUT_ELEMENT_DESC[numElements];
desc[0].SemanticName = "POSITION";
desc[0].SemanticIndex = 0;
desc[0].Format = DXGI_FORMAT_R32G32_FLOAT;
```

```
desc[0].InputSlot = 0;
desc[0].AlignedByteOffset = 0;
desc[0].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
desc[0].InstanceDataStepRate = 0;
desc[1].SemanticName = "COLOR";
desc[1].SemanticIndex = 0;
desc[1].Format = DXGI_FORMAT_R32G32B32A32_FLOAT;
desc[1].InputSlot = 0;
desc[1].AlignedByteOffset =
D3D11_APPEND_ALIGNED_ELEMENT;
desc[1].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
desc[1].InstanceDataStepRate = 0;
```

Далее воспользуемся функцией для создания описания входных данных

```
device->CreateInputLayout(desc, numElements, vertexShaderBuffer->GetBufferPointer(),vertexShaderBuffer->GetBufferSize(), &layout),
```

где layout – указатель на описание входных данных с типом ID3D11InputLayout*.

Перед вызовом функций Draw() или DrawIndexed() необходимо будет установить описание входных данных вершинного шейдера

```
deviceContext->IASetInputLayout(layout);
```

и непосредственно сам вершинный шейдер

```
deviceContext->VSSetShader(vertexShader, NULL, 0).
```

2.6. Пиксельные шейдеры

Пиксельный шейдер – еще один программируемый этап конвейера рендеринга. Только на этом этапе производится уже обработка отдельных пикселов выводимого треугольника. Входные данные пиксельного шейдера совпадают с выходными данными вершинного шейдера

```
struct INPUT
{
 float4 position : SV_POSITION;
 float4 color : COLOR0;
}
```

Выходные данные пиксельного шейдера – цвет пиксела. В нашем случае это тип float4 (четырехкомпонентный вектор RGBA). В общем случае у пиксельного шейдера может быть много выходных данных; кроме вывода цвета на поверхность рендеринга он может выводить данные в буфер глубины или на другие поверхности рендеринга одновременно.

Основная функция пиксельного шейдера будет выглядеть следующим образом:

```
float4 main(INPUT input) : SV_TARGET
{
    return input.color;
}
```

Пиксельный шейдер в этом примере достаточно простой. Он берет интерполированные данные цвета вершин и выводит их как цвет пиксела.

Процесс создания пиксельного шейдера эквивалентен созданию вершинного шейдера и изображен на рис. 2.8. Загрузка пиксельного шейдера также похожа на загрузку вершинного шейдера, за исключением того, что не надо указывать формат атрибутов вершин, поскольку данные поступают из вершинного шейдера и находятся в строго определенных регистрах, которые мы указываем в вершинном шейдере в структуре выходных данных и пиксельном шейдере в структуре входных данных.

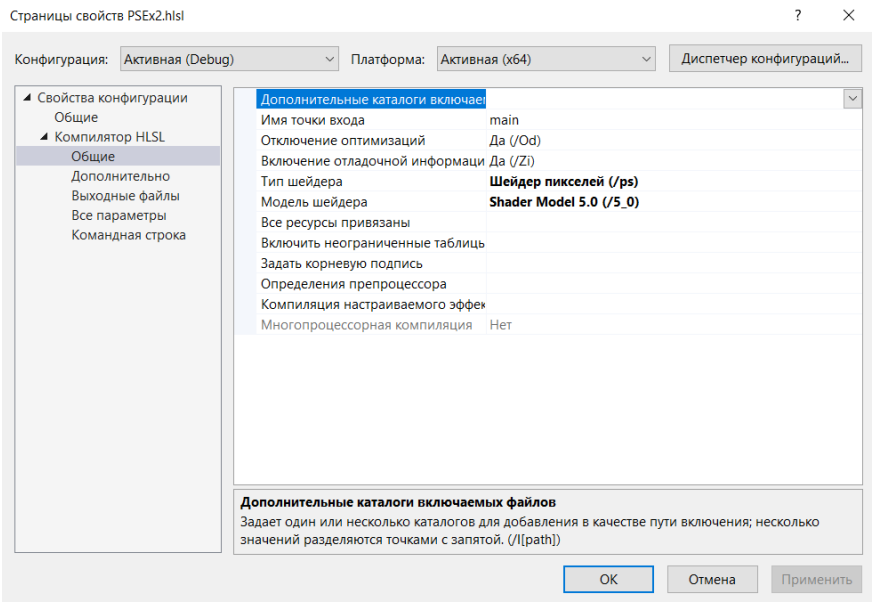


Рис. 2.8. Свойства пиксельного шейдера

Функция компиляции пиксельного шейдера

D3DCompileFromFile(name, NULL, NULL, "main", "ps_5_0", D3D10_SHADER_ENABLE_STRICTNESS, 0,&pixelShaderBuffer, &errorMessage),

где name – путь и имя файла с расширением .hlsl, а pixelShaderBuffer и errorMessage – указатели на область памяти с типом ID3D10Blob*, где будут находиться шейдер и сообщение об ошибке, если компиляция не удастся.

Функция загрузки скомпилированного шейдера

D3DReadFileToBlob(fname, &pixelShaderBuffer),

где name – путь и имя файла с расширением .cso.

После компиляции или загрузки воспользуемся функцией непосредственно создания пиксельного шейдера

device->CreatePixelShader(pixelShaderBuffer->GetBufferPointer(), pixelShaderBuffer->GetBufferSize(), NULL,&pixelShader),

где pixelShader – указатель типа ID3D11PixelShader* уже непосредственно на сам шейдер, которым и будем пользоваться в дальнейшем.

Перед вызовом функций Draw() или DrawIndexed() необходимо будет установить пиксельный шейдер

deviceContext->PSSetShader(pixelShader, NULL, 0).

На рис. 2.9 представлен пример вывода треугольника с интерполяцией цветов вершин. Пример можно скачать по ссылке: https://github.com/samoyow1973/Example2.

Рис. 2.9. Треугольник с интерполяцией цветов вершин

2.7. Геометрические шейдеры

Геометрический шейдер, или шейдер геометрии, – программируемый этап графического конвейера, который находится после вершинного шейдера. На вход геометрический шейдер принимает выходные данные вершинного шейдера, но есть возможность воспринимать эти данные как набор вершин, линий или треугольников, а на выход выдавать другой поток вершин, линий или треугольников. Причем, принимая, например, поток линий (по две вершины), на выходе геометрический шейдер способен генерировать поток треугольников. Другими словами, в геометрическом шейдере разработчик может создавать свою геометрию на основе входных данных. Пример входных данных геометрического шейдера (для одной вершины):

```
struct INPUT
{
	float4 position : SV_POSITION;
	float4 color : COLOR0;
	float thick : PSIZE0;
};
```

Выходные данные геометрического шейдера будут интерполированы и переданы на вход пиксельного шейдера. Так же как и в вершинном шейдере, выходными данными могут быть позиция, нормаль, цвет и многое другое, например:

```
struct OUTPUT
{
	float4 position : SV_POSITION;
	float4 color : COLOR0;
};
```

Рассмотрим основную функцию геометрического шейдера на примере рисования линий заданной толщины. Поскольку выводить будем список линий, на вход геометрического шейдера будем принимать две вершины – начало и конец отрезка. В геометрическом шейдере вместо отрезка будем выводить прямоугольник, состоящий из двух треугольников. Таким образом, на входе геометрический шейдер принимает две вершины, а на выходе возвращает шесть. Поэтому перед главной функцией необходимо объявить количество вершин, ко-

торые будут возвращены геометрическим шейдером ([maxvertexcount(6)]). Вся функция геометрического шейдера

```
[maxvertexcount(6)]
void main(line INPUT input[2], inout TriangleStream<OUTPUT> ThickLineStream)
{       OUTPUT output;
float3 vec = float3(input[0].position.xy - input[1].position.xy,0);
float3 up = float3(0, 0, 1.0);
float3 n = normalize(cross(vec, up));
float2 n1 = n.xy * input[0].thick;
float2 n2 = n.xy * input[1].thick;
output.position = float4(input[0].position.xy+n1,0,1.0);
output.color = input[0].color;
ThickLineStream.Append(output);
output.position = float4(input[1].position.xy + n2, 0, 1.0);
output.color = input[1].color;
ThickLineStream.Append(output);
output.position = float4(input[0].position.xy - n1, 0, 1.0);
output.color = input[0].color;
ThickLineStream.Append(output);
ThickLineStream.RestartStrip();
output.position = float4(input[0].position.xy - n1, 0, 1.0);
output.color = input[0].color;
ThickLineStream.Append(output);
output.position = float4(input[1].position.xy + n2, 0, 1.0);
output.color = input[1].color;
ThickLineStream.Append(output);
output.position = float4(input[1].position.xy - n2, 0, 1.0);
output.color = input[1].color;
ThickLineStream.Append(output);
ThickLineStream.RestartStrip();
}
```

Вершина добавляется в поток вывода функцией

Имя_потока.Append(выходные данные),

где Имя_потока – название потока, заданное пользователем в заголовке основной функции геометрического шейдера. Примитив можно вывести функцией Имя_потока.RestartStrip(). Необходимо понимать,

что треугольники из геометрического шейдера выводятся в типологии TRIANGLESTRIP, поэтому если требуется вывод TRIANGLELIST, то после добавления в поток вывода вершин каждого треугольника необходимо вызвать функцию RestartStrip().

Таким образом, можно предложить оптимизацию геометрического шейдера, потому что при выводе необходимы два треугольника, но второй треугольник использует две последние вершины первого треугольника. Для оптимизации необходимо изменить количество вершин, которые будут выведены ([maxvertexcount(6)]), убрать вывод четвертой и пятой вершин и совсем убрать вызовы ThickLineStream.RestartStrip().

Для вывода только контура треугольника из предыдущего параграфа необходимо заменить D3D11_PRIMITIVE_TOPOLOGY_ TRIANGLESTRIP на D3D11_PRIMITIVE_TOPOLOGY_LINESTRIP. Результат показан на рис. 2.10, причем линии такие тонкие, что их едва видно. Для вывода с помощью геометрического шейдера необходимо в формат вершины ввести третий элемент PSIZE (он будет определять ширину линии, выходящей из вершины)

```
UINT numElements;
numElements = 3;
//……......
// Здесь, например, расположено описание позиции вершины и ее цвета
//……......
desc[2].SemanticName = "PSIZE";
desc[2].SemanticIndex = 0;
desc[2].Format = DXGI_FORMAT_R32_FLOAT;
desc[2].InputSlot = 0;
desc[2].AlignedByteOffset =
D3D11_APPEND_ALIGNED_ELEMENT;
desc[2].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
desc[2].InstanceDataStepRate = 0;
```

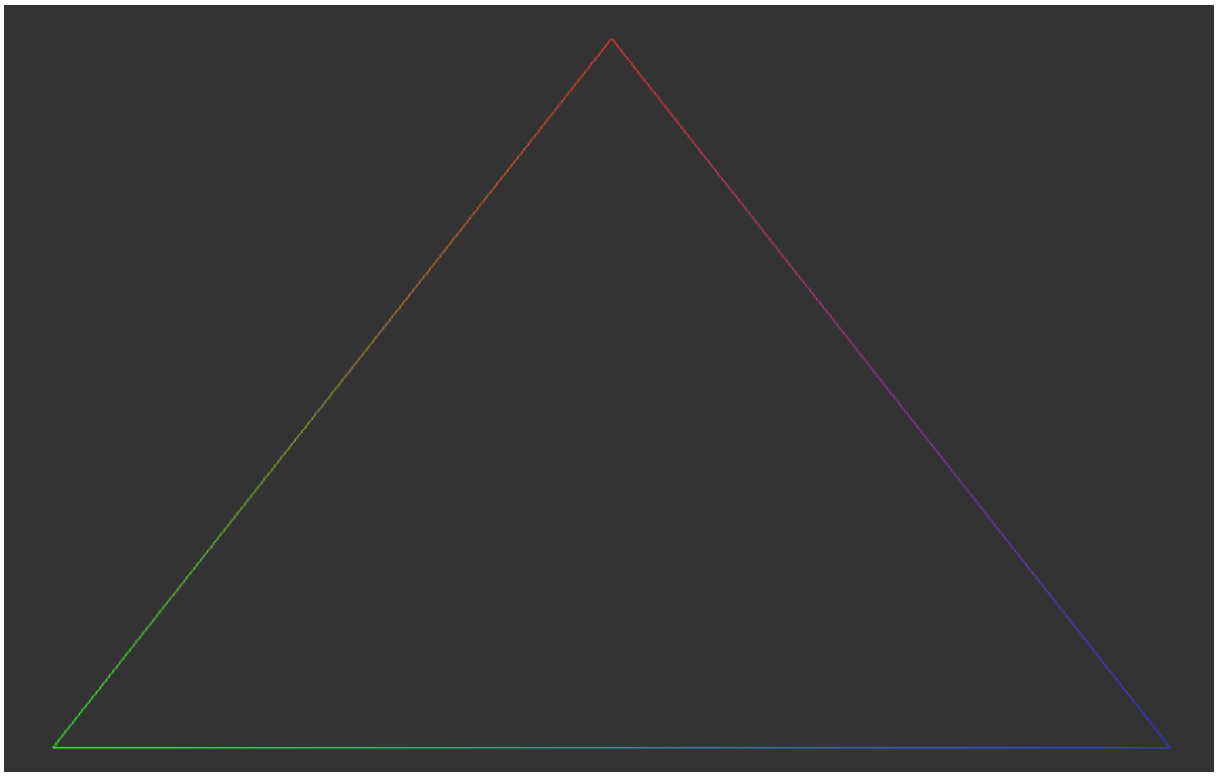


Рис. 2.10. Вывод линий с применением геометрического шейдера средствами DirectX

Кроме этого необходимо изменить структуры входных и выходных данных вершинного шейдера

```
struct INPUT
{
 float2 position : POSITION;
 float4 color : COLOR0;
 float thick : PSIZE0;};
struct OUTPUT{
 float4 position : SV_POSITION;
 float4 color : COLOR0;
 float thick : PSIZE0;
};
```

Остается не забыть передать в геометрический шейдер данные о толщине линии

```
output.thick = input.thick;
```

Процесс создания геометрического шейдера эквивалентен созданию пиксельного шейдера и изображен на рис. 2.11.

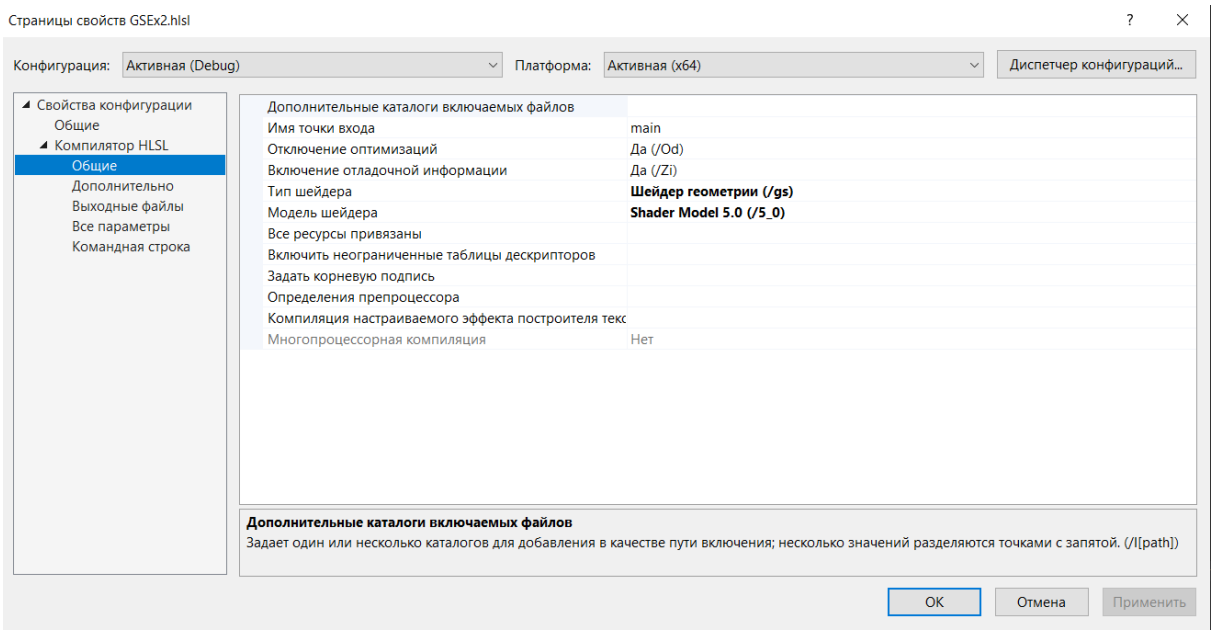


Рис. 2.11. Свойства геометрического шейдера

Функция компиляции геометрического шейдера

D3DCompileFromFile(name, NULL, NULL, "main", "gs_5_0", D3D10_SHADER_ENABLE_STRICTNESS, 0, &geometryShaderBuffer, &errorMessage),

где name – путь и имя файла с расширением .hlsl, а geometryShader-Buffer и errorMessage – указатели на область памяти с типом ID3D10Blob*, где будут находиться шейдер и сообщение об ошибке, если компиляция не удастся.

Функция загрузки скомпилированного шейдера

D3DReadFileToBlob(name, &geometryShaderBuffer),

где name – путь и имя файла с расширением .cso.

После компиляции или загрузки воспользуемся функцией непосредственно создания пиксельного шейдера

device->CreateGeometryShader(geometryShaderBuffer->GetBufferPointer(), geometryShaderBuffer->GetBufferSize(), NULL, &geometryShader,

где geometryShader – указатель типа ID3D11GeometryShader* уже непосредственно на сам шейдер, которым и будем пользоваться в дальнейшем.

Перед вызовом функций Draw() или DrawIndexed() необходимо будет установить пиксельный шейдер

deviceContext->GSSetShader(geometryShader, NULL, 0).

На рис. 2.12 показан пример использования геометрического шейдера для вывода линий заданной толщины. Таким образом, применение геометрического шейдера способно создавать собственную геометрию исходя из тех данных, которые поступают в геометрический конвейер. Пример можно скачать по ссылке: https://github.com/samoyow1973/Example11.

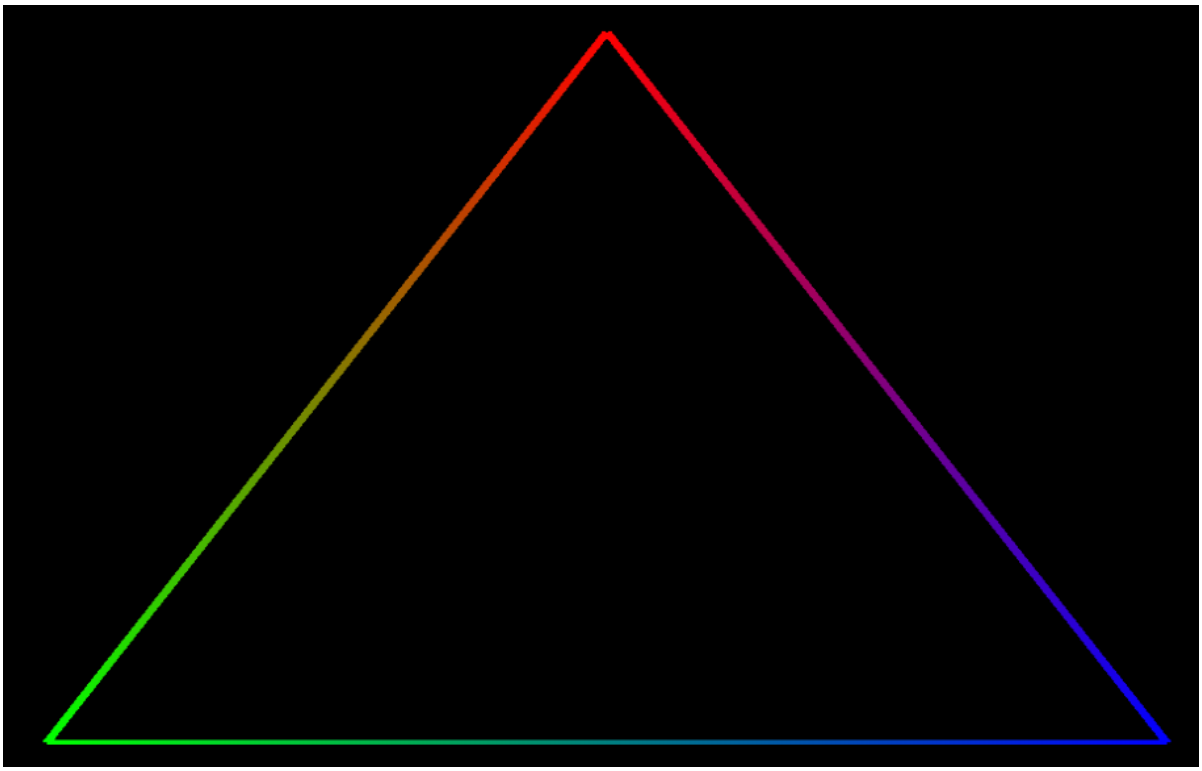


Рис. 2.12. Вывод линий заданной толщины с применением геометрического шейдера

2.8. Константные буферы

Константные буферы – это буферы данных, которые можно передавать в шейдеры. Это могут быть матрицы преобразований, параметры источников света, аргументы материалов и многое другое. Каждый тип шейдера может иметь до 15 константных буферов, в каждом из которых может находиться до 4096 констант. Необходимо отметить, что одна константа – это четырехкомпонентный вектор, а каждая компонента может иметь тип float, т. е. состоять из четырех байт. Это не говорит о том, что все данные должны иметь тип float. Они могут быть и целочисленными. Таким образом, размер одной константы 16 байт, а максимальный размер константного буфера 64 Кб. Рассмотрим создание константного буфера, заполнение его данными и загрузку в шейдер.

Создадим буфер по аналогии с буфером вершин или индексов. Количество констант определено переменной size, а данные, которыми необходимо заполнить константный буфер, располагаются по указателю data. Заполним необходимые поля в структурах

```
ID3D11Buffer* Buffer=NULL;
D3D11_BUFFER_DESC BufferDesc;
D3D11_SUBRESOURCE_DATA Data;
BufferDesc.Usage = D3D11_USAGE_DYNAMIC;
BufferDesc.ByteWidth = sizeof(XMFLOAT4) * size;
BufferDesc.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
BufferDesc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
BufferDesc.MiscFlags = 0;
BufferDesc.StructureByteStride = 0;
```

Data.pSysMem = data; // Указатель на массив данных, которые необходимо занести в константный буфер

```
Data.SysMemPitch = 0;
Data.SysMemSlicePitch = 0;
```

Далее создадим константный буфер, вызвав функцию

```
device->CreateBuffer(&BufferDesc, &Data, &Buffer).
```

Если при создании нет необходимости сразу заполнять константный буфер данными, то вместо второго параметра Data следует передать NULL. Обновить данные в константном буфере можно следующим образом:

```
D3D11_MAPPED_SUBRESOURCE mappedResource;
deviceContext->Map(Buffer, 0, D3D11_MAP_WRITE_DISCARD,
0, &mappedResource);
XMFLOAT4* temp=(XMFLOAT4*)mappedResource.pData;
// Переменная temp – указатель на константный буфер
// и по этому указателю можно копировать или заносить данные
deviceContext->Unmap(Buffer, 0);
```

Подключить константный буфер к шейдеру можно один или несколько раз даже внутри основного цикла вывода 3D-сцены. Для этого (для вершинного шейдера) используют функцию

```
deviceContext->VSSetConstantBuffers(number, count, &Buffer),
```

где number – номер слота, к которому подключают константные буферы; count – число константных буферов; &Buffer – массив кон-

стантных буферов. Функция позволяет подключить не только один, а сразу несколько константных буферов, начиная со слота number.

Для подключения константного буфера к пиксельному или геометрическому шейдеру служат аналогичные функции

deviceContext-> PSSetConstantBuffers (number, count, &Buffer),

deviceContext-> GSSetConstantBuffers (number, count, &Buffer).

В шейдере необходимо добавить несколько строк определения константного буфера

```
cbuffer NAME:register (b0) // нулевой слот константного буфера
{
 float4 constant0;
 float4 constant1;
 //...
 float4 constantn;
};
```

Доступ к элементам константного буфера можно получить, например, так:

```
float a= constant0.y;
float2 b= constant1.xy;
float4 c= constantn;
```

Пример создания константного буфера для передачи в шейдер времени с начала старта приложения:

```
ID3D11Buffer* CBuffer0 = NULL;
XMFLOAT4 data = XMFLOAT4(0,0,0,0);
D3D11_BUFFER_DESC BufferDesc;
D3D11_SUBRESOURCE_DATA Data;
BufferDesc.Usage = D3D11_USAGE_DYNAMIC;
BufferDesc.ByteWidth = sizeof(XMFLOAT4) * 1;
BufferDesc.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
BufferDesc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
BufferDesc.MiscFlags = 0;
BufferDesc.StructureByteStride = 0;
Data.pSysMem = &data;
Data.SysMemPitch = 0;
Data.SysMemSlicePitch = 0;
device->CreateBuffer(&BufferDesc, &Data, &CBuffer0);
```

Обновление буфера в основном цикле отрисовки и подключение его к вершинному шейдеру

```
// Заполним и установим константный буфер
D3D11_MAPPED_SUBRESOURCE mappedResource;
deviceContext->Map(CBuffer0, 0,
D3D11_MAP_WRITE_DISCARD, 0, &mappedResource);
XMFLOAT4* temp = (XMFLOAT4*)mappedResource.pData;
temp[0].x = time0/60.0f; // запишем текущее время в буфер
time0++;
deviceContext->Unmap(CBuffer0, 0);
deviceContext->VSSetConstantBuffers(0, 1, &CBuffer0);
```

Добавим определение константного буфера в вершинный шейдер

```
cbuffer TIME:register (b0)
{
 float4 time0;
};
```

И изменим основную функцию шейдера

```
OUTPUT main(INPUT input)
{
 OUTPUT output;
 output.position.xy = input.position;
 output.position.zw = float2(0.0, 1.0);
 float r = cos(time0.x + input.color.r * 3.14) * 0.5 + 0.5;
 float g = cos(0.5*time0.x + input.color.g * 3.14) * 0.5 + 0.5;
 float b = cos(0.25*time0.x + input.color.b * 3.14) * 0.5 + 0.5;
 output.color.rgb = float3(r,g,b);
 output.color.a = 1.0;
 return output;
}
```

Треугольник на экране начнет динамично менять цвета, как показано на рис. 2.13. Скачать пример можно по ссылке: https://github.com/samoyow1973/Example3.

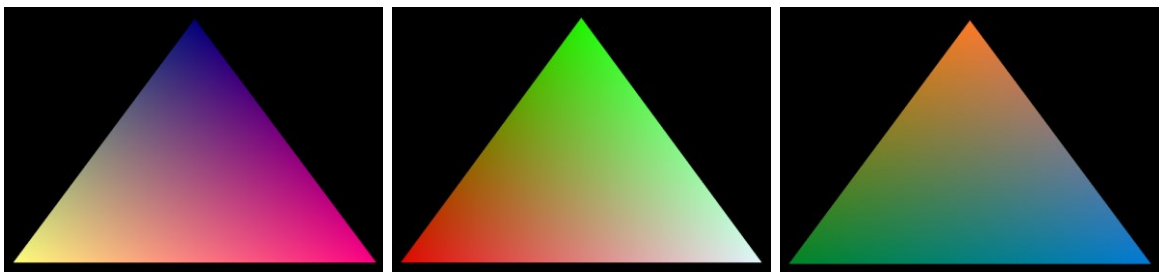


Рис. 2.13. Пример применения константного буфера

2.9. Текстурирование

В этом параграфе будут рассмотрены вопросы загрузки и наложения текстур. Для этого добавим в формат вершины текстурные координаты

```
struct VERTEX
{
    XMFLOAT2 pos;
    XMFLOAT4 color;
    XMFLOAT2 tex;
};
```

Создадим описание формата вершины как входных данных вершинного шейдера

```
D3D11_INPUT_ELEMENT_DESC* desc = NULL;
UINT numElements;
numElements = 3;
desc = new D3D11_INPUT_ELEMENT_DESC[numElements];
desc[0].SemanticName = "POSITION";
desc[0].SemanticIndex = 0;
desc[0].Format = DXGI_FORMAT_R32G32_FLOAT;
desc[0].InputSlot = 0;
desc[0].AlignedByteOffset = 0;
desc[0].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
desc[0].InstanceDataStepRate = 0;
desc[1].SemanticName = "COLOR";
desc[1].SemanticIndex = 0;
desc[1].Format = DXGI_FORMAT_R32G32B32A32_FLOAT;
desc[1].InputSlot = 0;
desc[1].AlignedByteOffset =
D3D11_APPEND_ALIGNED_ELEMENT;
desc[1].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
desc[1].InstanceDataStepRate = 0;
desc[2].SemanticName = "TEXCOORD";
desc[2].SemanticIndex = 0;
desc[2].Format = DXGI_FORMAT_R32G32_FLOAT;
desc[2].InputSlot = 0;
desc[2].AlignedByteOffset =
D3D11_APPEND_ALIGNED_ELEMENT;
```

```
desc[2].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
desc[2].InstanceDataStepRate = 0;
```

Изменим стурктуру входных и выходных данных в вершинном шейдере

```
struct INPUT
{
 float2 position : POSITION;
 float4 color : COLOR0;
 float2 tex:TEXCOORD0;
};
struct OUTPUT
{
 float4 position : SV_POSITION;
 float4 color : COLOR0;
 float2 tex:TEXCOORD0;
};
```

Теперь основная функция вершинного шейдера выглядит следующим образом:

```
OUTPUT main(INPUT input)
{
 OUTPUT output;
 output.position.xy = input.position;
 output.position.zw = float2(0.0, 1.0);
 float r = cos(time0.x + input.color.r * 3.14) * 0.5 + 0.5;
 float g = cos(0.5*time0.x + input.color.g * 3.14) * 0.5 + 0.5;
 float b = cos(0.25*time0.x + input.color.b * 3.14) * 0.5 + 0.5;
 output.color.rgb = float3(r,g,b);
 output.color.a = 1.0;
 output.tex = input.tex;
 return output;
}
```

Добавим еще одну вершину в вершинный буфер и один индекс в индексный буфер. Используя топологию вывода треугольников D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP, передадим в графический конвейер четыре вершины и нарисуем два треугольника, которые будут образовывать прямоугольник, как показано на рис. 2.14.

Рис. 2.14. Прямоугольник

Пользуясь тем, что у вершин теперь есть текстурные координаты, значения которых в верхнем левом углу составляют (0; 0), а в правом нижнем углу (1; 1) и линейно интерполированы по всему прямоугольнику, изменим пиксельный шейдер следующим образом:

```
struct INPUT
{
 float4 position : SV_POSITION;
 float4 color : COLOR0;
 float2 tex:TEXCOORD0;
};
float4 main(INPUT input) : SV_TARGET
{
 float x = input.tex.x-0.5;
 float y = input.tex.y-0.5;
 float r = x * x + y * y;
 if (r > 0.25) discard;
 return input.color;
}
```

Оператор discard позволяет отбрасывать и не выводить пикселы на экран, которые не удовлетворяют какому-либо условию. С помощью интерполированных текстурных координат посчитаем расстояния от центра прямоугольника и отбросим те пикселы, которые боль-

ше некоторой константы. В результате получим эллипс (вообще, это круг, но с учетом соотношений сторон окна он превращается в эллипс), как показано на рис. 2.15.

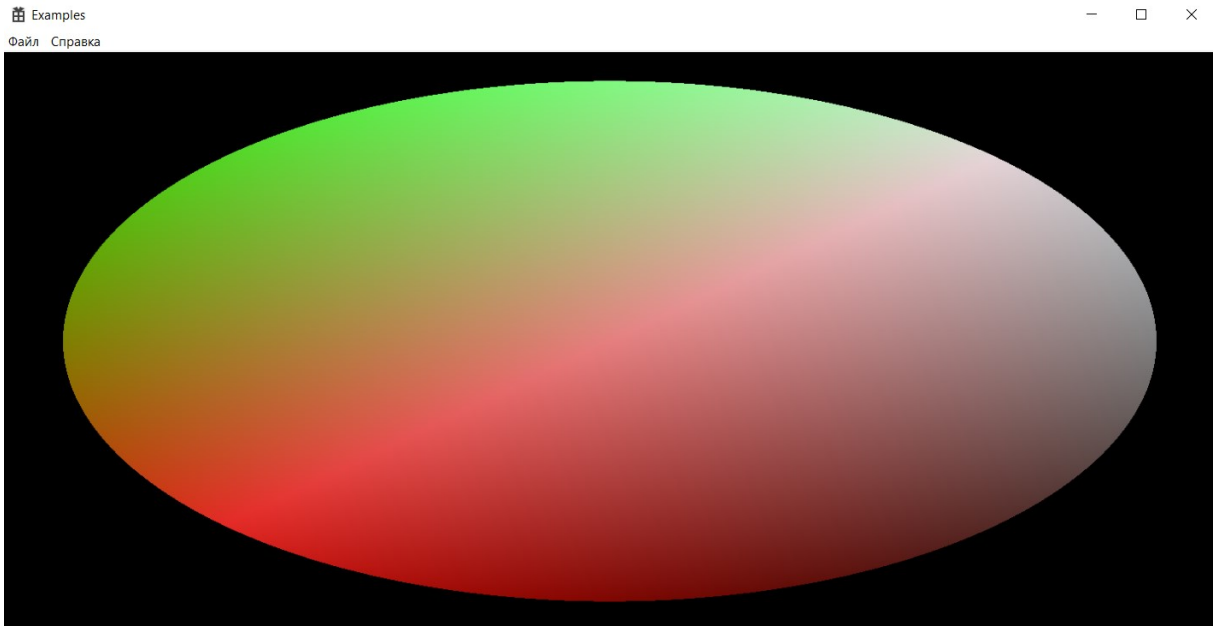


Рис. 2.15. Эллипс, реализуемый в пиксельном шейдере

Теперь необходимо установить настройки стадии текстурирования, заполнив структуру D3D11_SAMPLER_DESC, создав и установив состояние текстурирования, как было показано в п. 2.4,

```
D3D11_SAMPLER_DESC descs;
descs.Filter = D3D11_FILTER_MIN_MAG_MIP_LINEAR;
descs.AddressU = D3D11_TEXTURE_ADDRESS_WRAP;
descs.AddressV = D3D11_TEXTURE_ADDRESS_WRAP;
descs.AddressW = D3D11_TEXTURE_ADDRESS_WRAP;
descs.MipLODBias = 0.0f;
descs.MaxAnisotropy = 1;
descs.ComparisonFunc = D3D11_COMPARISON_ALWAYS;
descs.BorderColor[0] = 0;
descs.BorderColor[1] = 0;
descs.BorderColor[2] = 0;
descs.BorderColor[3] = 0;
descs.MinLOD = 0;
descs.MaxLOD = D3D11_FLOAT32_MAX;
device->CreateSamplerState(&descs, &Samplerstate);
```

Рассмотрим загрузку текстуры из файла. Для загрузки текстур формата dds воспользуемся следующим кодом (необходимо подключение библиотек DirectXTK и DirectXTEX):

```
TexMetadata imageMetadata;
ScratchImage* image = new ScratchImage();
LoadFromDDSFile(filename, DDS_FLAGS_NONE, &imageMetadata, *image);
CreateShaderResourceView(device, image->GetImages(), image->GetImageCount(), imageMetadata, &texture);
```

где filename – путь и имя файла; texture – указатель на текстуру с типом ID3D11ShaderResourceView*.

Перед вызовом функций Draw() или DrawIndexed() необходимо установить текстуру функцией

```
deviceContext->PSSetShaderResources(number, count, &texture),
```

где number – начальный номер слота загрузки текстур; count – число устанавливаемых текстур (максимально их может быть 128); &texture – адрес массива текстур.

Таким образом, можно установить одну текстуру в нужный слот или установить сразу массив текстур.

Пиксельный шейдер при этом примет вид

```
Texture2D Texture: register(t0);
SamplerState SampleType: register(s0);
struct INPUT
{
 float4 position : SV_POSITION;
 float4 color : COLOR0;
 float2 tex:TEXCOORD0;
};
float4 main(INPUT input) : SV_TARGET
{ return float4(Texture.Sample(SampleType, input.tex).rgb, 1.0);}
```

Следует обратить внимание на то, что SampleType связан с регистром s0, куда ранее функцией deviceContext->PSSetSamplers(0, 1, &Samplerstate) было установлено созданное состояние текстурирования. Результат текстурирования приведен на рис. 2.16.

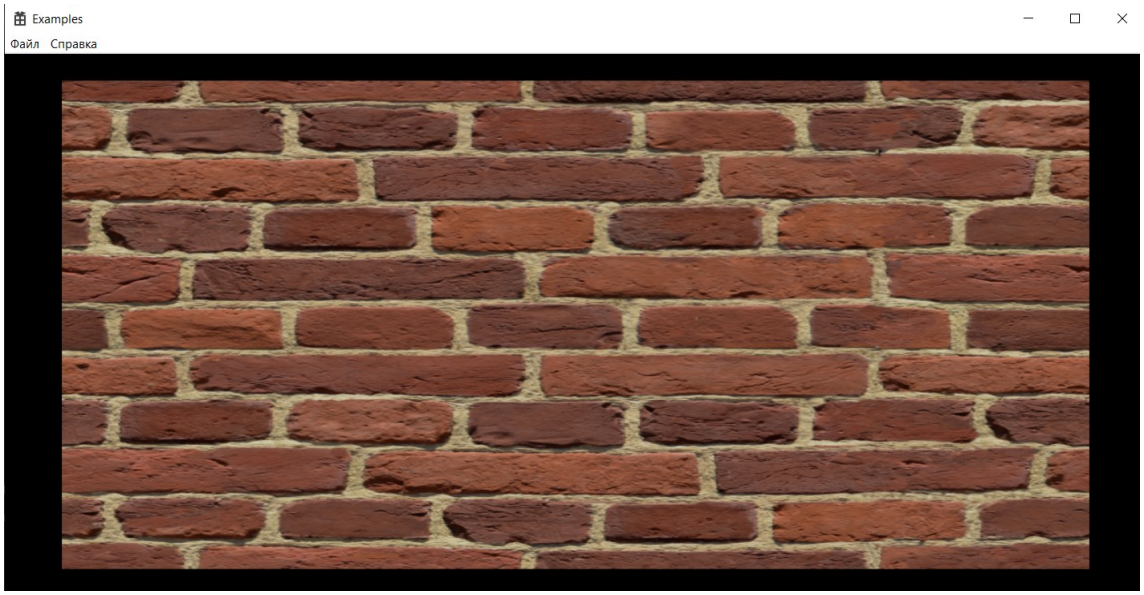


Рис. 2.16. Наложение текстуры

Взяв функцию main пиксельного шейдера, приведенную в начале параграфа, и изменив последнюю строку на

return input.color*float4(Texture.Sample(SampleType, input.tex).rgb, 1.0);

получим результат, показанный на рис. 2.17. В этом примере цвет пикселов текстуры перемножается с цветом прямоугольника. Пример можно скачать по ссылке: https://github.com/samoyow1973/Example4.

Рис. 2.17. Текстурирование и цвет

Попробуйте самостоятельно получить результат как на рис. 2.18, где граница объекта является волнообразной и меняется с течением времени.

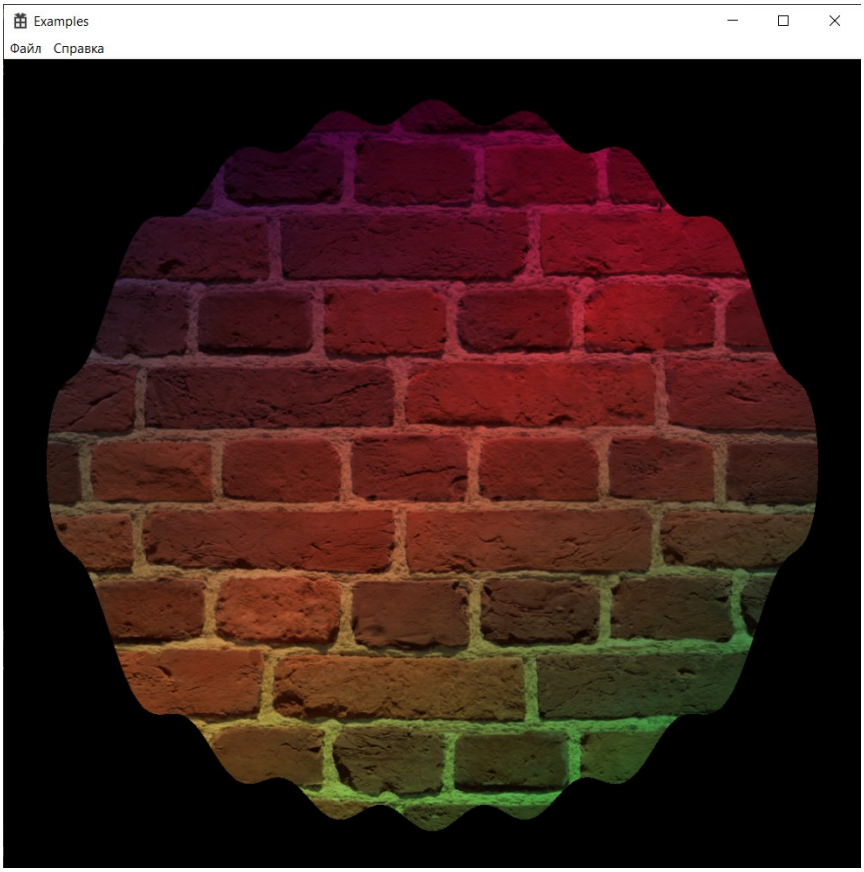


Рис. 2.18. Самостоятельное задание

2.10. Отображение трехмерных объектов

В предыдущих параграфах отображались только двумерные объекты. Такие объекты обычно используют для рисования линий, меню, текста и различных спрайтов. Теперь рассмотрим отображение трехмерных объектов. Как было показано в п. 1.6, прежде чем трехмерный объект будет отображен на двумерном экране, каждая его вершина должна претерпеть некоторые изменения. Таким образом, координаты вершины должны домножиться на матрицу проекции, матрицу камеры и матрицу трансформации самого объекта (мировую матрицу). Обычно (но не всегда) используют перспективную проекцию. Матрицу перспективной проекции можно создать, применив функцию

XMFLOAT4X4 project;

XMStoreFloat4x4(&project,XMMatrixPerspectiveFovLH(angle, aspect, nearZ, farZ)),

где angle – угол взгляда по вертикали; aspect – соотношение сторон окна или экрана; nearZ – расстояние от камеры до ближней плоскости отсечения; farZ – расстояние от камеры до дальней плоскости отсечения.

Матрицу вида можно создать, используя следующую функцию:

XMFLOAT4X4 view;

XMStoreFloat4x4(&view, XMMatrixLookAtLH(XMLoadFloat3(&pos), XMLoadFloat3(&at) + XMLoadFloat3(&pos), XMLoadFloat3(&up))),

где pos – трехкомпонентный вектор позиции камеры; at – трехкомпонентный вектор направления камеры; up – трехкомпонентный вектор, перпендикулярный вектору at и показывающий вертикальную составляющую камеры.

Теоретически матрицу проекции и матрицу вида надо передать в виде констант в вершинный шейдер перед выводом 3D-сцены, но поскольку эти матрицы все равно будут перемножены, целесообразно передавать в вершинный шейдер их произведение

XMFLOAT4X4 view_project;

XMStoreFloat4x4(&view_project,

XMMatrixMultiply (XMLoadFloat4x4(&view), XMLoadFloat4x4(&project))).

Создадим вершинный буфер, описывающий куб из 24 вершин (по четыре вершины на каждую грань) и индексный буфер на 36 индексов (по три на каждый треугольник). Укажем у вершин каждой грани текстурные координаты и индивидуальный цвет. Топологию треугольников будем выводить списком

D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST.

Каждая вершина будет иметь следующую структуру:

```
struct VERTEX
{
    XMFLOAT3 pos; // трехкомпонентный вектор позиции вершины
    XMFLOAT4 color; // цвет вершины
    XMFLOAT2 tex; // текстурные координаты
};
```

Например, вершины и индексы одной грани задаются следующим образом:

```
Triangle[0].pos.x = 0.5f;
Triangle[0].pos.y = 0.5f;
Triangle[0].pos.z = 0.5f;
Triangle[0].color = XMFLOAT4(1.0f, 0.0f, 0.0f, 1.0f);
Triangle[0].tex.x = 0.0;
Triangle[0].tex.y = 0.0;
Triangle[1].pos.x = 0.5f;
Triangle[1].pos.y = -0.5f;
Triangle[1].pos.z = 0.5f;
Triangle[1].color = XMFLOAT4(1.0f, 0.0f, 0.0f, 1.0f);
Triangle[1].tex.x = 0.0;
Triangle[1].tex.y = 1.0;
Triangle[2].pos.x = -0.5f;
Triangle[2].pos.y = -0.5f;
Triangle[2].pos.z = 0.5f;
Triangle[2].color = XMFLOAT4(1.0f, 0.0f, 0.0f, 1.0f);
Triangle[2].tex.x = 1.0;
Triangle[2].tex.y = 1.0;
Triangle[3].pos.x = -0.5f;
Triangle[3].pos.y = 0.5f;
Triangle[3].pos.z = 0.5f;
Triangle[3].color = XMFLOAT4(1.0f, 0.0f, 0.0f, 1.0f);
Triangle[3].tex.x = 1.0;
Triangle[3].tex.y = 0.0;
Indices[0] = 0; Indices[1] = 1; Indices[2] = 2;
Indices[3] = 2; Indices[4] = 3; Indices[5] = 0;
```

В описании входных данных вершинного шейдера укажем формат позиции вершины как: DXGI_FORMAT_R32G32B32_FLOAT.

Поскольку грани будут загораживать друг друга, необходимо включить z-буфер, установив в структуре

D3D11_DEPTH_STENCIL_DESC поле DepthEnable = TRUE.

Константный буфер создадим на девять констант. Одна константа – время, еще по четыре будут занимать матрицы (каждая матрица – 16 компонентов float) view_project и world.

Вершинный шейдер примет следующий вид:

```
buffer CBUFFER0:register (b0)
{
 float4 time0;
 float4x4 vpmatrix;
 float4x4 wmatrix;
};
struct INPUT
{
 float3 position : POSITION;
 float4 color : COLOR0;
 float2 tex:TEXCOORD0;
};
struct OUTPUT
{
 float4 position : SV_POSITION;
 float4 color : COLOR0;
 float2 tex:TEXCOORD0;
};
OUTPUT main(INPUT input)
{
 OUTPUT output;
 float4x4 mvp = mul(vpmatrix, wmatrix);
 output.position = mul(mvp, float4(input.position, 1.0));
 output.color = input.color;
 output.tex = input.tex;
 return output;
}
```

А пиксельный шейдер (без наложения текстур) будет выглядеть следующим образом:

```
Texture2D Texture: register(t0);
SamplerState SampleType: register(s0);
cbuffer TIME:register (b0)
```

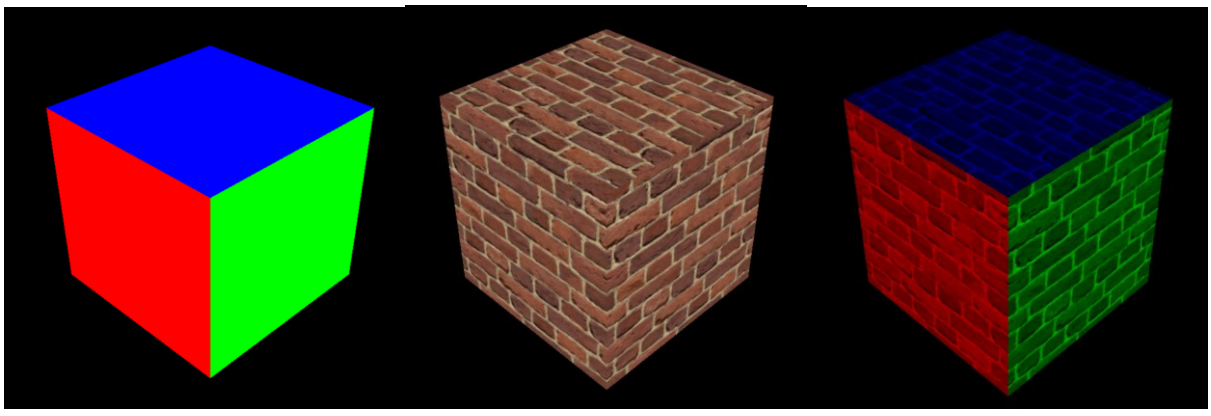
```
{
 float4 time0;
};
struct INPUT
{
 float4 position : SV_POSITION;
 float4 color : COLOR0;
 float2 tex:TEXCOORD0;
};
float4 main(INPUT input) : SV_TARGET
{
 return input.color;
}
```

В результате можно наблюдать разноцветный куб, как показано на рис. 2.19, *а*. Если применить текстурирование, как было показано в предыдущем параграфе, результат будет выглядеть как на рис. 2.19, *б*.

Изменив главную функцию пиксельного шейдера

```
return input.color*float4(Texture.Sample(SampleType,
input.tex).rgb,1.0);
```

можно смешать цвет грани и текстурирование, как показано на рис. 2.19, *в*.



а) *б)* *в)*

Рис. 2.19. Трехмерный куб

Добавив в основной цикл отрисовки (перед заполнением константного буфера) строки

```
XMMATRIX world;
XMMATRIX wy = XMMatrixRotationY(time0 / 60.0f * XM_PIDIV4);
XMMATRIX wx = XMMatrixRotationX(time0 / 60.0f * XM_PIDIV4 / 2.0f);
XMMATRIX wz = XMMatrixRotationZ(time0 / 60.0f * XM_PIDIV4 / 3.0f);
 world = XMMatrixMultiply(wy, wx);
 world = XMMatrixMultiply(world, wz);
```

заключающиеся в вычислении матрицы поворота объекта вокруг осей *Y*, *X*, *Z* с разной скоростью, получим в результате вращающийся куб, как показано на рис. 2.20. Пример можно скачать по ссылке: https://github.com/samoyow1973/Example5.

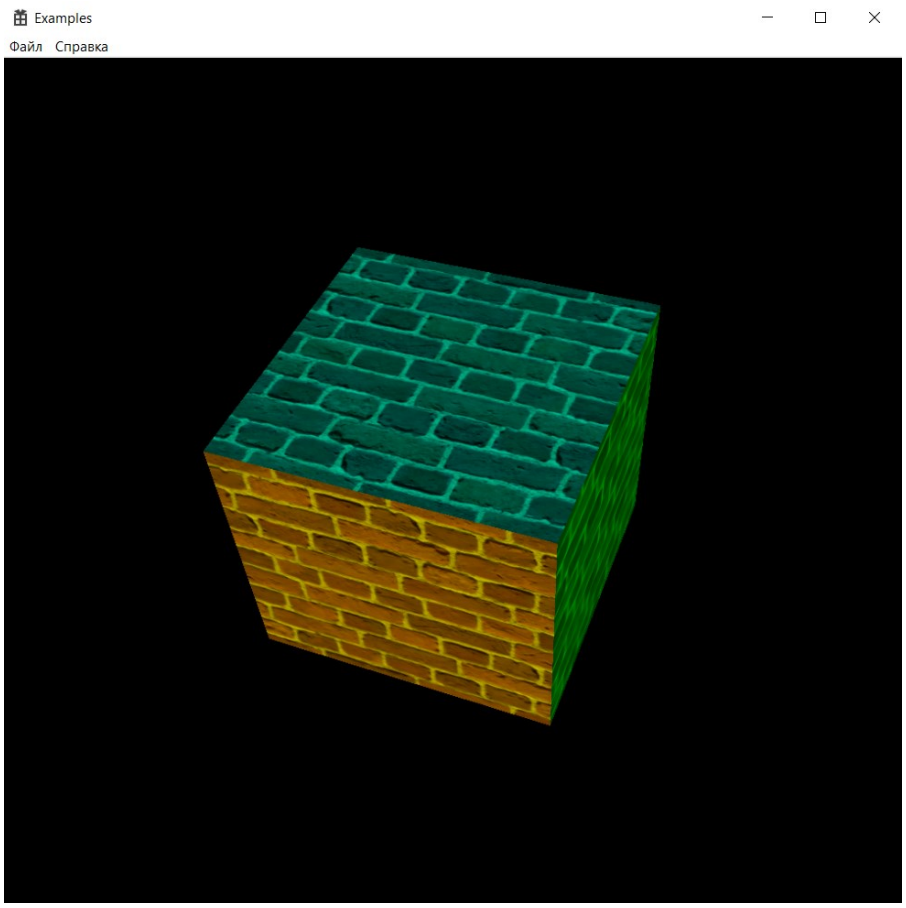


Рис. 2.20. Вращение 3D-куба

Как можно заметить, текстурированный куб на рис. 2.19, *б* выглядит несколько безжизненным. Связано это с тем, что в реализованных примерах отсутствует один из основных аспектов трехмерной графики – освещение. Именно источникам света и моделям освещения и будет посвящена следующая глава.

2.11. Отображение прозрачных объектов

Отображение прозрачных объектов сопряжено с рядом сложностей. Во-первых, прозрачные объекты должны быть выведены после того, как выведены все непрозрачные объекты. Во-вторых, выводить прозрачные объекты необходимо от дальних к ближним, чтобы результат получался правдоподобным.

В этом параграфе не будут рассмотрены алгоритмы сортировки прозрачных объектов по дальности от камеры. Ограничимся только необходимыми техниками.

Сначала создадим два состояния смешивания, заполнив структуру D3D11_BLEND_DESC для прозрачных и непрозрачных объектов,

```
D3D11_BLEND_DESC descb;
descb.AlphaToCoverageEnable = FALSE;
descb.IndependentBlendEnable = FALSE;
descb.RenderTarget[0].BlendEnable = FALSE; // отключить смешивание
descb.RenderTarget[0].SrcBlend = D3D11_BLEND_SRC_ALPHA;
descb.RenderTarget[0].DestBlend =
D3D11_BLEND_INV_SRC_ALPHA;
descb.RenderTarget[0].BlendOp = D3D11_BLEND_OP_ADD;
descb.RenderTarget[0].SrcBlendAlpha = D3D11_BLEND_ONE;
descb.RenderTarget[0].DestBlendAlpha = D3D11_BLEND_ZERO;
descb.RenderTarget[0].BlendOpAlpha=D3D11_BLEND_OP_ADD;
descb.RenderTarget[0].RenderTargetWriteMask = 15;
device->CreateBlendState(&descb, &BlendstateAlphaOff);
descb.RenderTarget[0].BlendEnable = TRUE; // включить смешивание
device->CreateBlendState(&descb, &BlendstateAlphaOn);
```

Необходимо отметить что существует достаточно много различных вариантов смешивания пиксела, который будет выведен на поверхность отображения, и пиксела, который уже есть на этой поверхности. Подробно обо всех вариантах можно прочитать в Windows DirectX Graphics Documentation, которая устанавливается вместе с DirectX SDK.

В нашем примере смешивание происходит по формуле

$$color = alpha \cdot scr + (1 - alpha) \cdot dest,$$

где *color* – результатирующий цвет пиксела; *scr* – цвет пиксела, который выводим на поверхность отображения; *dest* – цвет пиксела, который уже есть на этой поверхности; *alpha* – альфа-компонент цвета вершин полупрозрачного объекта.

Для вывода полупрозрачной плоскости добавим еще четыре вершины в вершинный буфер, четыре индекса – в индексный буфер и еще одну текстуру полупрозрачного объекта. В основном цикле отрисовки установим режим непрозрачности

```
float blendFactor[4]; // Установить фактор смешивания
blendFactor[0] = 0.0f;
blendFactor[1] = 0.0f;
blendFactor[2] = 0.0f;
blendFactor[3] = 0.0f;
deviceContext->OMSetBlendState(BlendstateAlphaOff, blendFactor, 0xffffffff);
```

и отобразим вращающийся куб как в предыдущем примере. Затем установим единичную матрицу преобразования объекта, обновим и установим константный буфер, текстуру полупрозрачного объекта, режим полупрозрачности

```
deviceContext->OMSetBlendState(BlendstateAlphaOn, blendFactor, 0xffffffff);
```

Далее выведем полупрозрачный четырехугольник. В результате получим трехмерную сцену, изображенную на рис. 2.21, состоящую из динамически двигающегося непрозрачного куба и статического полупрозрачного прямоугольника. Пример можно скачать по ссылке: https://github.com/samoyow1973/Example6.

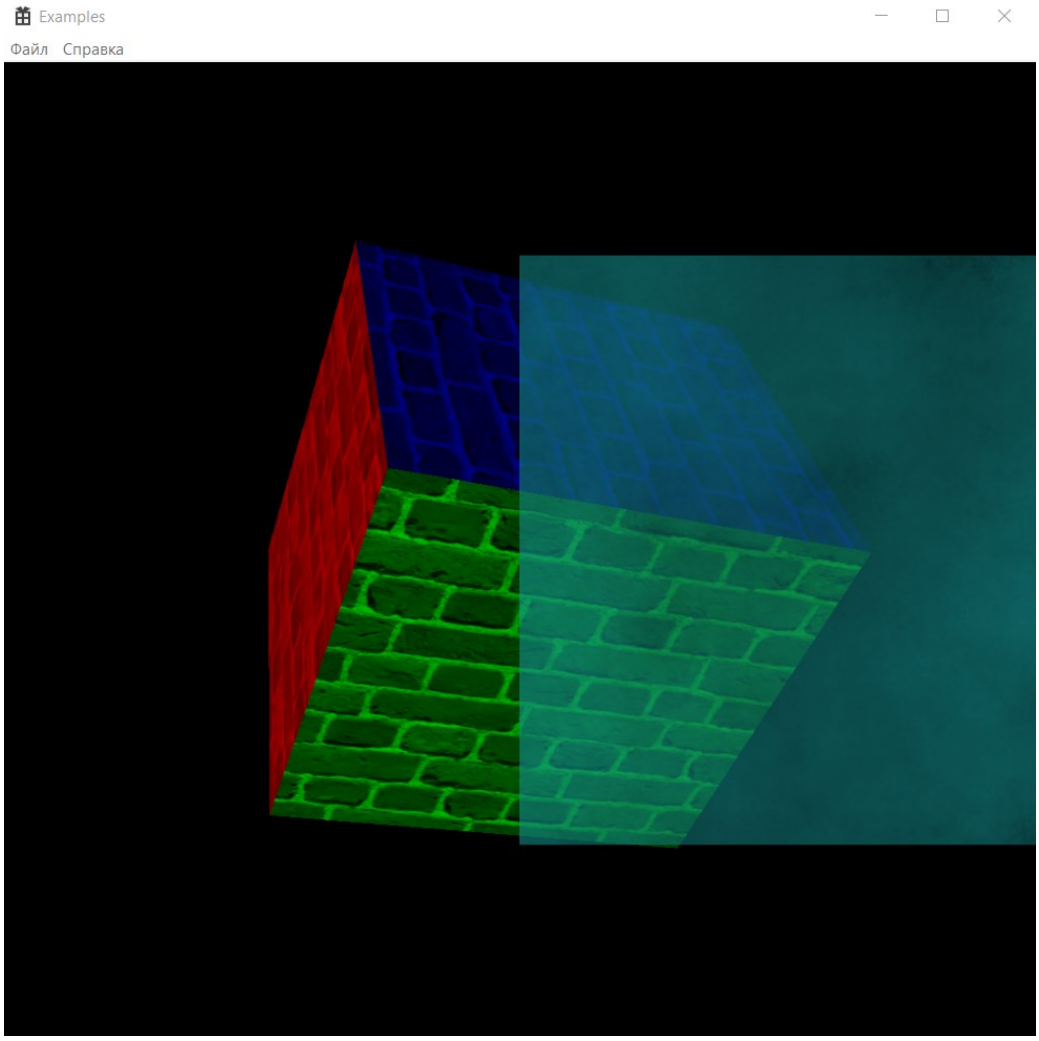


Рис. 2.21. Пример полупрозрачного объекта

Контрольные вопросы

1. Что такое задний буфер?
2. Что такое вертикальная синхронизация?
3. Каковы основные этапы функции инициализации?
4. Что хранится в вершинном буфере?
5. Для чего используют индексный буфер?
6. В чем отличие топологий LIST и STRIP?
7. Как управлять состояниями этапов конвейера рендеринга?
8. Как включить или выключить буфер глубины?
9. Как отключить запись в буфер глубины, но оставить тест глубины?
10. Как отключить вывод на поверхность рендеринга, но оставить вывод в буфер глубины?

11. Как запретить вывод «обратных» сторон треугольников?
12. Какой язык программирования шейдеров используется в DirectX?
13. В чем отличие вершинных шейдеров от фрагментных?
14. Как загрузить и скомпилировать шейдер?
15. Зачем нужно описание формата вершинных данных?
16. Какие данные получает вершинный шейдер?
17. Какие данные получает фрагментный шейдер?
18. Какие данные возращаются на выходе фрагментного шейдера?
19. Зачем нужен геометрический шейдер?
20. Какие данные поступают на вход геометрического шейдера?
21. Какие данные возращаются на выход шеометрического шейдера?
22. Что такое константный буфер?
23. Каков максимальный размер константного буфера?
24. Что такое текстурные координаты?
25. Как загрузить текстуру?
26. Как назначить текстуру для наложения на полигоны?
27. Каков максимальный размер текстурного буфера?
28. Что значит фильтрация текстур?
29. Как установить необходимый фильтр для текстур?
30. Что такое смешивание?
31. Как установить нужный режим смешивания?
32. Какова основная формула наложения полупрозрачных объектов?

Глава 3. ОСВЕЩЕНИЕ

3.1. Типы источников света

Освещение – одна из важнейших составляющих трехмерной графики. Именно от того, насколько качественно сделано освещение в программе, и будет зависеть реалистичность картинки в представленной трехмерной сцене. Однако до сих пор еще не создана модель освещения, достаточно близкая к реальной физической модели. На сегодняшний день наиболее достоверная модель освещения – модель на основе трассировки лучей (Ray Trasing). Но она требует достаточно мощных вычислительных ресурсов видеокарты, поэтому в данной главе будут рассмотрены более простые модели освещения.

Прежде чем рассматривать модели освещения, надо определиться с тем, какие типы источников света встречаются и как можно классифицировать свет вообще. Принято считать, что свет бывает трех видов:

– *рассеянный* (ambient), т. е. такой, который равномерно распределен по всей трехмерной сцене, не имеет направления, а имеет только трехкомпонентную составляющую RGB и равномерно освещает все объекты без исключения с одинаковой интенсивностью;

– *диффузный* (diffuse) – этот свет кроме цветовой компоненты (она зависит от материала объекта, который его испускает) имеет направление, и интенсивность его снижается пропорционально квадрату расстояния от источника;

– *отраженный* (specular) (бликовый свет, цветовая компонента которого зависит только от источника света) – имеет направление, и его интенсивность зависит как от объекта, от которого он отразился, так и от расстояния.

Сами же источники света в трехмерной сцене можно классифицировать следующим образом:

– *источники рассеянного света* (ambient) – т. е. все объекты в трехмерной сцене будут равномерно им освещены;

– *источники направленного света* – такие источники, где свет имеет направление и диффузную составляющую, но не имеет координат в пространстве и все его лучи параллельны. Интенсивность такого

источника остается постоянной во всей трехмерной сцене. Это аналог очень удаленных источников света, например солнца;

– *точечные источники* – такие источники света имеют диффузную составляющую, не имеют направления, но имеют положение в пространстве. Наиболее близкий аналог – лампочка на потолке без абажура;

– *точечные конусные источники* (как показано на рис. 3.1) – они кроме диффузной составляющей имеют и положение в пространстве, и направление освещения, и угол конуса угла освещения, и внутренний угол, где интенсивность освещения постоянна и степень снижения интенсивности освещения – от внутреннего угла к внешнему. Ближайший аналог – фонарик.

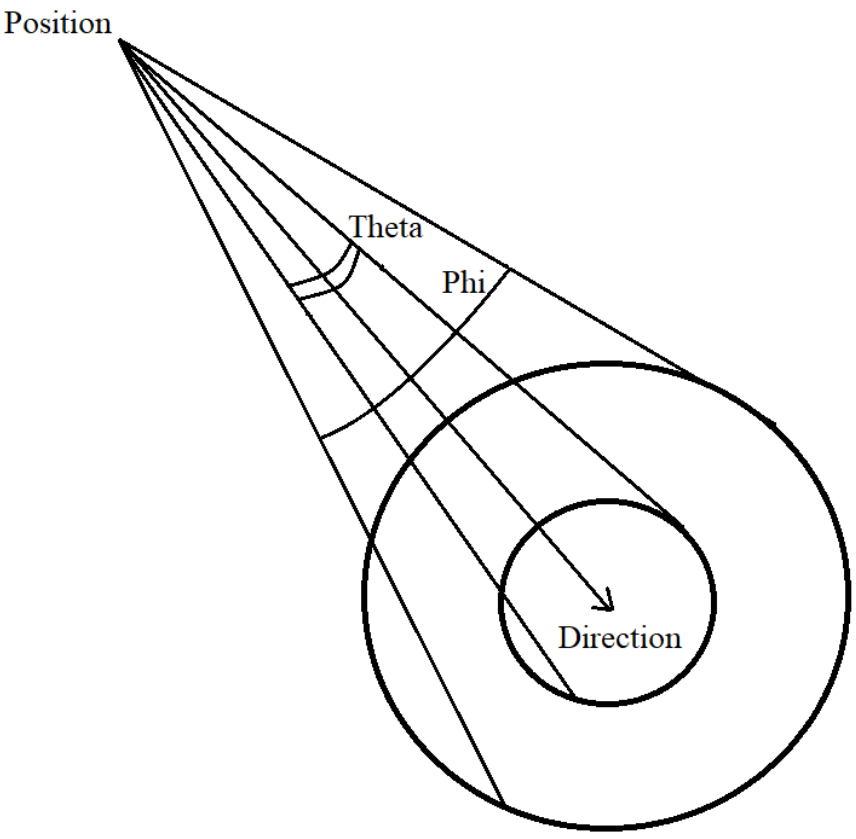


Рис. 3.1. Точечный конусный источник света

Таким образом, при программировании освещения потребуется структура источников света, которая должна содержать тип источника, диффузную составляющую источника света, положение в пространстве, направление источника света и характеристики для конусного источника света

```
struct LIGHT
{
	XMFLOAT4 color; // RGBA
	XMFLOAT4 position; // позиция
	XMFLOAT4 direction; // нормализованный вектор направления
	XMFLOAT4 parametrs; // тип источника, phi, theta, falloff
};
```

Именно эти параметры источника света необходимо передавать в шейдер при отображении трехмерного объекта. В случае с единственным источником света в сцене это не вызывает сложностей, но при большом количестве источников света в сцене необходимо будет находить другие решения. Кроме того, в шейдер будем передавать позицию камеры, это будет нужно при расчете бликов.

Таким образом, структура константного буфера выглядит следующим образом:

```
cbuffer CBUFFER0:register (b0)
{
 float4 time0; // время
 float4x4 vpmatrix; // произведение матрицы проекции и матрицы вида
 float4x4 wmatrix; // матрица трансформации объекта
 float4 l_color; // цвет источника света
 float4 l_pos; // позиция источника света
 float4 l_dir; // направление источника света
 float4 l_param; // параметры источника света
 float4 view_pos; // позиция камеры
};
```

Следует отметить, что для реализации какой-либо модели освещения потребуется такой атрибут вершины, как нормаль (перпендикуляр к касательной плоскости).

Структура вершины, таким образом, изменится

```
struct VERTEX
{
	XMFLOAT3 pos;
	XMFLOAT3 normal;
	XMFLOAT4 color;
	XMFLOAT2 tex;
};
```

Потребуется внести изменения в описание входных данных вершинного шейдера

```
numElements = 4;
…
desc[1].SemanticName = "NORMAL";
desc[1].SemanticIndex = 0;
desc[1].Format = DXGI_FORMAT_R32G32B32_FLOAT;
desc[1].InputSlot = 0;
desc[1].AlignedByteOffset =
D3D11_APPEND_ALIGNED_ELEMENT;
desc[1].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
desc[1].InstanceDataStepRate = 0;
```

3.2. Затенение по Гуро

Этот метод затенения был предложен А. Гуро в 1971 году. Метод заключается в вычислении цвета каждой вершины сообразно нормали вершины. А для всех точек треугольника эти значения интерполируются. Поскольку вычисления достаточно простые и происходят в вершинном шейдере, затенение по Гуро – наиболее производительный метод, хотя и не лишен недостатков [1].

Вершинный шейдер при этом будет

```
cbuffer CBUFFER0:register (b0)
{
 float4 time0;
 float4x4 vpmatrix;
 float4x4 wmatrix;
 float4 l_color;
 float4 l_pos;
 float4 l_dir;
 float4 l_param;
 float4 view_pos;
};
struct INPUT
{
 float3 position : POSITION;
 float3 normal : NORMAL;
 float4 color : COLOR0;
```

```
 float2 tex : TEXCOORD0;
};
struct OUTPUT
{
 float4 position : SV_POSITION;
 float3 normal : NORMAL;
 float4 color : COLOR0;
 float2 tex:TEXCOORD0;
};
OUTPUT main(INPUT input)
{
 OUTPUT output;
 float4x4 mvp = mul(vpmatrix, wmatrix);
 output.position = mul(mvp, float4(input.position, 1.0));
 output.normal = normalize(mul(wmatrix, float4(input.normal,
0.0)).xyz);
 float3 vpos= mul(wmatrix, float4(input.position, 1.0)).xyz;
 // Расчет диффузной составляющей
 float diffuse = saturate(-dot(output.normal, l_dir.xyz));
 // Расчет отраженной составляющей
 float p = 10; // степень глянцевости
 float ks = 0.0; // коэффициент зеркального блика
 float3 s_color = float3(1,1,1); // цвет блика
 float3 V = normalize(view_pos.xyz - vpos);
 float3 R = normalize(reflect(l_dir.xyz, output.normal));
 float specular=0;
 if(diffuse >0)
  specular = pow(max(dot(V, R), 0.0), p);

 output.color.rgb = input.color.rgb * diffuse + s_color*specular*ks;
 output.color.a = input.color.a;
 output.tex = input.tex;
 return output;}
```

В этом шейдере расчет диффузной составляющей осуществляется за счет вычисления скалярного произведения между обратным направлением источника света и нормалью вершины. Отрицательные результаты отбрасываются. Расчет отраженной составляющей заключается в вычислении отраженного вектора света и скалярного произ-

ведения между вектором взгляда на вершину и отраженным вектором. Результат возводится в некоторую степень, показывающую степень глянцевости поверхности. При точечном источнике света в шейдере вместо строки

float3 R = normalize(reflect(l_dir.xyz, output.normal));

необходимо вставить строки

float3 L = vpos - l_pos.xyz;

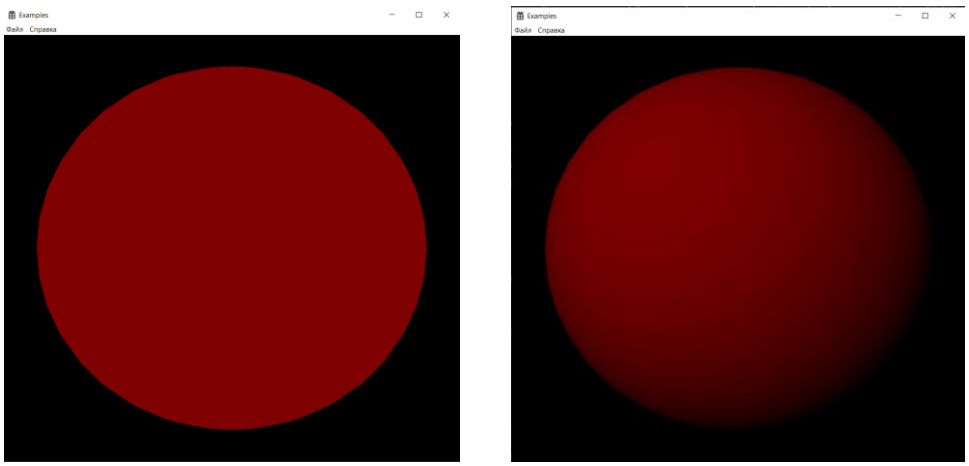
float3 R = normalize(reflect(L, output.normal));

Это связано с тем, что направление света при точечном источнике рассчитывается как разница между вершиной объекта и позицией источника света.

Пиксельный шейдер при этом выглядит следующим образом:

```
struct INPUT
{
 float4 position : SV_POSITION;
 float3 normal : NORMAL;
 float4 color : COLOR0;
 float2 tex:TEXCOORD0;
};
float4 main(INPUT input) : SV_TARGET
{
return input.color;
}
```

На рис. 3.2, *а* представлен вид шара без освещения, а на рис. 3.2, *б* вид с затенением по Гуро при направленном источнике света и при отсутствии бликов.



а) *б)*

Рис. 3.2. Затенение по Гуро

На рис. 3.3 показано затенение по Гуро с бликами с разной степенью отражений от поверхности.

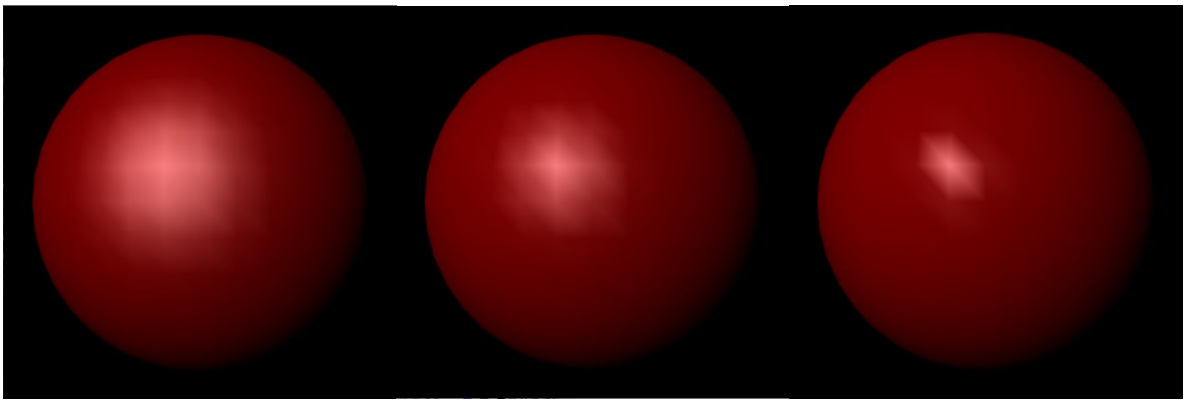


Рис. 3.3. Блики при затенении по Гуро

Как можно видеть, блики при затенении по Гуро выглядят скорее как искры, чем как блики. Связано это с линейной интерполяцией бликовой составляющей. Кроме того, поскольку происходит интерполция рассчитанного цвета вершины, а не нормали, диффузная составляющая затенения может быть отображена неверно. Эти проблемы были решены в модели освещенности Фонга, о чем пойдет речь в следующем параграфе. Пример можно скачать по ссылке: https://github.com/samoyow1973/Example7.

3.3. Затенение по Фонгу

Этот метод затенения был предложен вьетнамским ученым-информатиком Б. Фонгом в 1975 году. Метод затенения Фонга отличается от затенения по Гуро интерполяцией не цветов вершин, рассчитанных в вершинном шейдере, а интерполяцией необходимых векторов: нормалей, вектора направления света, вектора взгляда [5; 6]. Основные расчеты диффузной и зеркальной составляющих света происходят в пиксельном шейдере. При этом из вершинного шейдера необходимо передать рассчитанные для вершин векторы. Структура выходных данных вершинного шейдера

```
struct OUTPUT
{
 float4 position : SV_POSITION;
 float3 normal : NORMAL;
```

```
 float2 tex:TEXCOORD0;
 float3 V:TEXCOORD1;
 float3 L:TEXCOORD2;
};
```

Цвет вершины в данном шейдере не передаем, потому что в примере пиксельного шейдера цвет поверхности будет задан напрямую. В реальных программах за цвет поверхности отвечает такая характеристика трехмерных объектов, как материал, но о материалах речь пойдет в следующих параграфах.

Вычисление необходимых векторов в вершинном шейдере не вызывает серьезных сложностей

```
output.normal = normalize(mul(wmatrix, float4(input.normal, 0.0)).xyz);
output.V = normalize(view_pos.xyz - vpos);
output.L = normalize(vpos- l_pos.xyz); // для точечного источника
// output.L= l_dir.xyz; // для направленного источника
```

В пиксельном шейдере сначала производим нормализацию всех векторов, потому что их длины могли измениться при интерполяции,

```
float3 N = normalize(input.normal);
float3 L = normalize(input.L);
float3 V = normalize(input.V);
```

Затем производим расчет диффузной составляющей пиксела

```
float3 diffuse = max(dot(N, -L), 0.0) * diffuse_albedo;
```

где diffuse_albedo – это отражаемая объектом часть света, т. е. цвет объекта.

Далее рассчитываем бликовую составляющую

```
float3 R = reflect(L, N);
float3 specular = pow(max(dot(R, V), 0.0), specular_power) * specular_albedo;,
```

где specular_power – степень глянцевости объекта; specular_albedo – степень глянцевости бликов.

Необходимо отметить еще одну составляющую цвета пикселов, это составляющая для рассеянного освещения – ambient. Результатирующий цвет пикселя вычисляется как сумма всех составляющих

```
return float4(ambient+diffuse+specular,1.0);
```

Результаты затенения по Фонгу приведены на рис. 3.4 для разных степеней глянцевости.

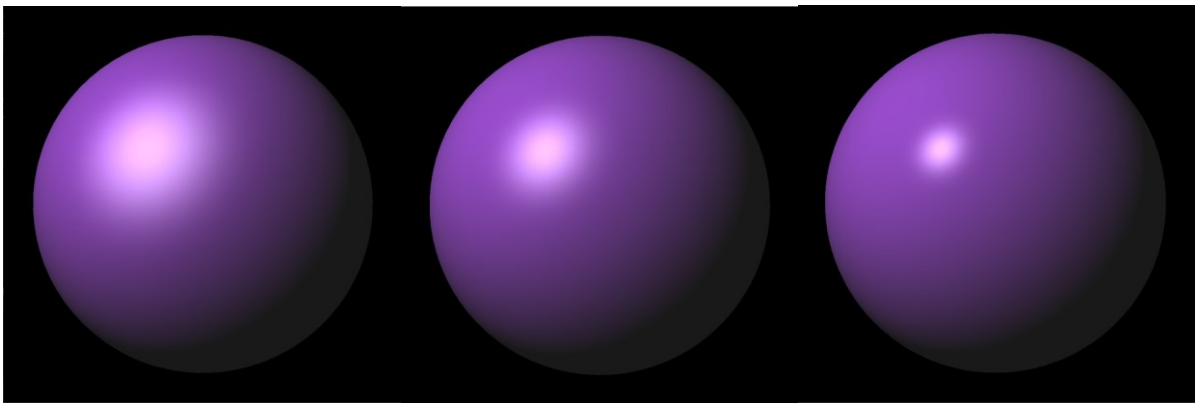


Рис. 3.4. Затенение по Фонгу

С точки зрения вычислительных ресурсов затенение по Фонгу требует больших затрат, поскольку основная часть вычислений ведется не в вершинном, а в пиксельном шейдере, и не для трех вершин, а для всех пикселов треугольника. Особенно «дорогим» оказывается расчет отраженного вектора *R*.

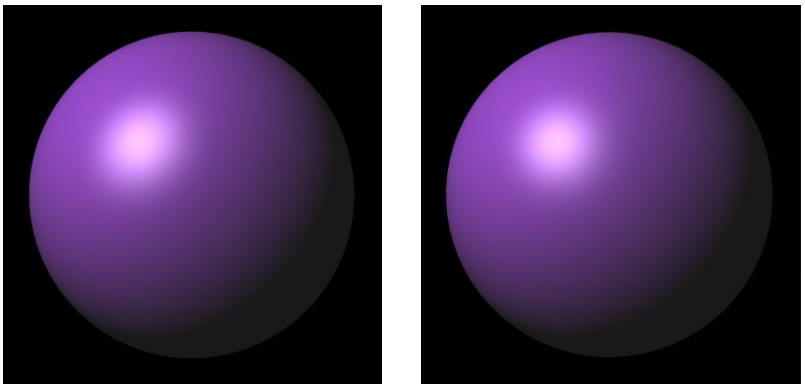
Чтобы снизить вычислительную нагрузку и повысить быстродействие, была предложена аппроксимация отраженного вектора вектором половинного пути (среднее между векторами света и взгляда). Данный метод затенения называют *методом затенения Блинна – Фонга*.

Расчет бликовой составляющей в этом случае будет выглядеть так:

```
float3 H = normalize(V-L);
float3 specular = pow(max(dot(H, N), 0.0), specular_power*4) *
specular_albedo;
```

Однако при использовании вектора половины пути интенсивность зеркального отражения становится выше, вследствие чего необходимо увеличить степень глянцевости в четыре раза. На рис. 3.5, *а* приведено затенение по Фонгу, а на рис. 3.5, *б*, – по Блинну – Фонгу.

Для шара в виде трехмерного объекта разница практически незаметна.



а) *б)*

Рис. 3.5. Затенение: а – по Фонгу; б – по Блинну – Фонгу

В прил. 2 приведены полные листинги вершинного и пиксельного шейдеров для затенения по Фонгу и по Блинну – Фонгу. Пример можно скачать по ссылке: https://github.com/samoyow1973/Example8.

3.4. Физически корректное освещение

Физически корректный рендеринг (physically-based rendering), или PBR, – это набор техник визуализации, которые с высокой степенью согласуются с теорией распространения света. Поскольку PBR базируется на физически достоверной модели, то такой подход даст более реалистичную картинку, нежели затенение по Фонгу или по Блинну – Фонгу. Основное преимущество заключается в том, что создаваемые материалы (имеются в виду различные коэффициенты трехмерных поверхностей) будут выглядеть так, как они задумывались художниками, независимо от условий освещения [7; 8].

Итак, чтобы получить физически корректный рендеринг, необходимо, во-первых, произвести гамма-коррекцию. Связано это с тем, что если взять белый цвет (1; 1; 1) и умножить его на 0,5, то человеческий глаз должен воспринимать результат как (0,73; 0,73; 0,73), а не как (0,5; 0,5; 0,5). Во-вторых, физически корректный рендеринг должен основываться на модели отражающих микрограней. В-третьих, необходимо соблюдать закон сохранения энергии. В-четвертых, PBR должен использовать двулучевую функцию отражающей способности BRDF (Bidirectional reflectance distribution function).

Для осуществления гамма-коррекции достаточно после всех вычислений (а производиться они должны в линейном цветовом пространстве) видоизменить цвет каждого пиксела по формуле

$$Color = clr^{1/2.2},$$

где *clr* – рассчитанный цвет пиксела; *Color* – скорректированный цвет.

Эти операции можно выполнить в самом конце пиксельного шейдера PBR

```
color = pow(color, 1.0/2.2);
```

Модель отражающих микрограней основывается на представлении о том, что каждая поверхность состоит из микроскопических зеркал. Чем поверхность более шероховатая, тем больше будет рассеиваться отраженный свет и более широким будет зеркальный блик. Вводя коэффициент шероховатости, можно считать что некоторая доля микрограней ориентирована перпендикулярно некоторому вектору *H*, который является средним вектором между вектором направления света и вектором взгляда наблюдателя (аналогично затенению по Блинну – Фонгу). Чем больше микрограней ориентировано вдоль среднего вектора *H*, тем более ярким и четким будет блик. Коэффициент шероховатости лежит в пределах от 0 до 1, показывая долю ориентированных микрограней. Чем этот коэффициент ближе к нулю, тем больше доля ориентированных микрограней и тем четче блики.

Сохранение энергии надо понимать следующим образом. Часть света зеркально отразится (блики), а поскольку коэффициент шероховатости не выходит за пределы единицы, то зеркально отраженный свет не превысит падающего. Действительно, чем больше площадь блика, тем он более тусклый. Но кроме отраженного света есть еще и преломленный, который проникает в материал, фотоны света теряют свою энергию, сталкиваясь с другими частицами, и либо эта энергия иссякнет, либо часть фотонов покинет поверхность в случайных направлениях. Сделаем предположение, что в рассматриваемой модели PBR весь преломленный свет поглощается и рассеивается на небольшом участке поверхности, игнорируя тот факт, что свет может покидать поверхность и вне этой небольшой области. Существуют модели PBR, учитывающие эффект подповерхностного рассеивания, но они ресурсоемкие и выходят за рамки настоящего учебного пособия.

Дополнительно следует сказать, что рассмотренный закон сохранения энергии не относится к металлам, а относится только к диэлектрикам. Металлы по-другому взаимодействуют со светом и не рассевают его, а только отражают. Таким образом, металлические поверхности не имеют диффузного света. И эту особенность необходимо учитывать в модели PBR.

Таким образом, исходя из закона сохранения энергии энергия преломленного (поглощенного материалом) света есть энергия падающего света за вычетом энергии зеркально отраженного.

Двулучевая модель BRDF на сегодняшний день в большинстве графических движков подчиняется модели Кука – Торренса [6; 7]. Эта модель, в отличие от модели Фонга, обеспечивает выполнение закона сохранения энергии. При этом вершинный шейдер остается без изменений, как и при затенении по Фонгу. Вершинный шейдер необходим для расчета и последующей интерполяции векторов нормалей, взгляда и направления на источник света.

Согласно модели Кука – Торренса зеркальная доля отраженного света вычисляется по формуле

$$K_{spec} = \frac{DFG}{4(\vec{V}\vec{N})(\vec{N}\vec{L})}, \quad (1)$$

где D – функция распределения отраженного света с учетом микрограней; G – функция распределения самозатенения и самоперекрытия; F – коэффициент отражения Френеля, данная функция показывает долю отраженного света; $\vec{V}$ – вектор взгляда; $\vec{L}$ – вектор направления света; $\vec{N}$ – нормаль к поверхности объекта.

Функции распределения D и G являются приближенными. Существуют различные приближения, одними из них являются функции *GGX*, разработанные Брюсом Вальтером.

Распределение функции самозатенения и перекрытия использует метод Смита, разработанный в 1967 году. Смысл метода заключается в подсчетах потерь света от источника света до поверхности и от поверхности до наблюдателя. Поэтому функцию разбивают на две части и считают потери света от угла между $\vec{L}$ и $\vec{N}$ и от угла между $\vec{V}$ и $\vec{N}$.

Функция распределения G

$$G(v,m) = \chi(v,m)\frac{2}{1+\sqrt{1+\alpha^2\tan(\theta)^2}},$$

где $\chi()$ – функция, возвращающая 1 или 0 в зависимости от совпадения направления векторов света и нормали; α – коэффициент шероховатости; θ – угол между вектором нормали и вектором света в одном случае и вектором взгляда в другом. В пиксельном шейдере функция *G* будет выглядеть следующим образом:

```
float GGX_G (float cosTheta, float alpha)
{
 float cosTheta_sqr = saturate(cosTheta*cosTheta);
 float tan2 = ( 1 - cosTheta_sqr ) / cosTheta_sqr;
 float GP = 2 / ( 1 + sqrt( 1 + alpha * alpha * tan2 ) );
 return GP;
}
```

В качестве одного из параметров в функцию GGX передаем результат скалярного произведения векторов нормали с вектором света или взгляда, поскольку этот результат – косинус угла между векторами. Таким образом, функция *G* будет выглядеть как

```
float roug_sqr = roughness*roughness;
float G = GGX_G (dot(N,V), roug_sqr) * GGX_G (dot(N,L),
roug_sqr);
```

На рис. 3.6 выведены значения функции *G* при освещении шара источником света слева от него и при коэффициентах шероховатости от 0,2 до 1.

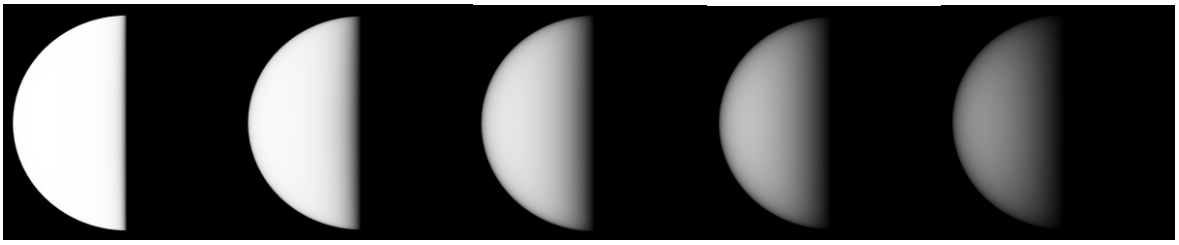


Рис. 3.6. Значение функции G при разных коэффициентах шероховатости

GGX-распределение функции *D* микрограней вычисляют по формуле

$$D(m) = \frac{\alpha^2 \chi(m,n)}{\pi \cos(\theta)^4 (\alpha^2 + \tan(\theta)^2)^2},$$

где $\chi()$ – функция, возвращающая 1 или 0 в зависимости от совпадения направления векторов света и нормали; α – квадрат коэффициента

шероховатости; θ – угол между вектором нормали и вектором H – половинным вектором между векторами света и взгляда.

Заменяя тангенс угла его выражением через косинус, получим, что в пиксельном шейдере функция D будет выглядеть следующим образом:

```
float GGX_D (float cosTheta, float alpha)
{
 float alpha2 = alpha * alpha;
 float NH_sqr = saturate(cosTheta * cosTheta);
 float den = NH_sqr * alpha2 + (1.0 - NH_sqr);
 return alpha2 / ( PI * den * den );
}
```

Вызов этой функции будет осуществляться как

```
float D = GGX_Distribution(dot(N,H), roughness*roughness);
```

На рис. 3.7 выведены значения функции D при освещении шара источником света слева от него при коэффициентах шероховатости от 0,2 до 1 и добавленной гамма-коррекции. Необходимо заметить, что если все сделано верно, то при коэффициенте шероховатости, равном единице, половина шара однотонно закрашивается значениями (153; 153; 153) ±1.

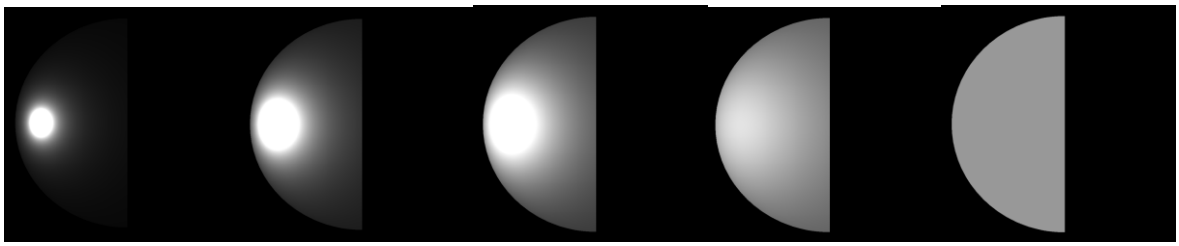


Рис. 3.7. Значение функции D при разных коэффициентах шероховатости

Вычисление коэффициентов Френеля – достаточно ресурсоемкая операция. Однако, используя аппроксимацию Шлика, можно воспользоваться следующей формулой:

$$R(\theta) = R_0 + (1 - R_0)(1 - \cos(\theta))^5$$

$$R_0 = \left(\frac{n1 - n2}{n1 + n2}\right)^2,$$

где R_0 – отношение коэффициентов преломления; θ – угол между векторами падения света и нормалью.

В шейдере функция вычисления коэффициентов Френеля будет выглядеть следующим образом:

```
float3 FresnelSchlick(float3 F0, float cosTheta)
{
 return F0 + (1.0 - F0) * pow(1.0 - saturate(cosTheta), 5.0);
}
```

Необходимо обратить внимание, что F_0 имеет размерность трехмерного вектора, т. е. RGB, потому что коэффициенты отражения Френеля могут быть разными для разных каналов. Вызвать эту функцию можно следующим кодом:

```
float3=FresnelSchlick(F0, H*V);
```

Теперь вся функция вычисления отраженного света по формуле (1) выглядит следующим образом:

```
float3 specK = G*D*F*0.25/(N*V);
```

Деление на скалярное произведение *NL* отсутствует, поскольку в конечном итоге все равно придется умножать на *NL* и это значение сократится. На рис. 3.8 представлены результаты для различных коэффициентов шероховатости и для коэффициентов Френеля: (0,04; 0,04; 0,04), (0,4; 0,4; 0,4), (1,0; 0,86; 0,56).

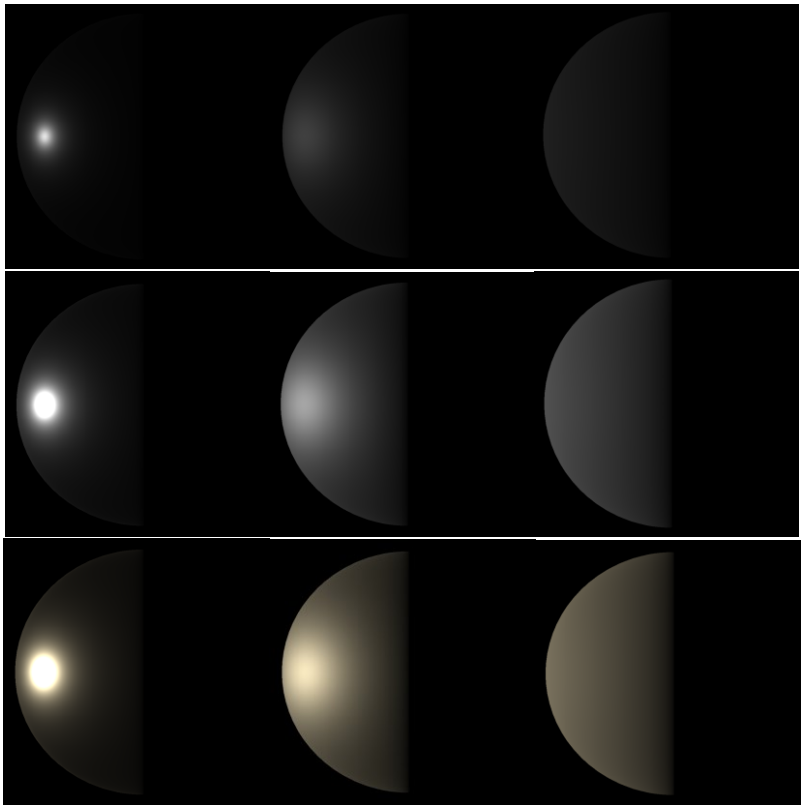


Рис. 3.8. Отраженный свет в модели Кука – Торренса

После вычисления доли отраженного света доля рассеянного света составит 1 – FresnelSchlick(). Этот свет пройдет через поверхность и в конечном итоге будет частично поглощен, а частично переизлучен. Переизлучение будет происходить в разных направлениях, т. е. случайным образом по полусфере. Данное рассеивание описывается распределением Ламберта

$$diffuse = lightcolor * albedo * \vec{NL} / \pi ,$$

где *lightcolor* – цвет источника света; *albedo* – цвет, который способен переизлучать объект.

В пиксельном шейдере это выразится кодом

```
float3 diffK = saturate(1.0-F)*albedo*NL/PI;
```

Результатирующий цвет пиксела и его гамма-коррекция будут рассчитаны как

```
float3 color = diffK+specK;
color = pow(color, 1.0 / 2.2);
return float4(color,1.0);
```

На рис. 3.9 приведено освещение шаров как на рис. 3.8, но с учетом диффузной, т. е. рассеиваемой, составляющей. В качестве albedo выбирались значения (0,0; 0,2; 0,7), (1,0; 0,1; 0,0), (0,04; 0,04; 0,04).

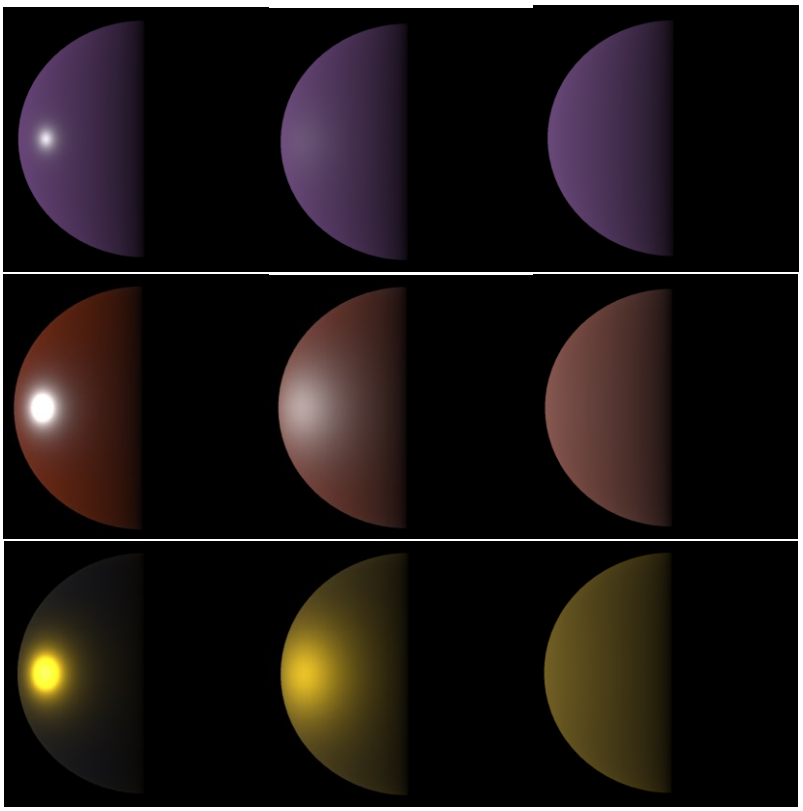


Рис. 3.9. Отраженный и рассеянный свет в модели Кука – Торренса

Теперь осталось разобраться с вопросом металлов и диэлектриков. При вычислении коэффициентов Френеля используется такой параметр, как F_0. F_0 – это степень отражения поверхности при нулевом угле падения света. Значение F_0 меняется в зависимости от материала и приобретает цветной оттенок для металлов. Что касается диэлектриков, то они выглядят достаточно достоверно при значениях F_0 = (0,04; 0,04; 0,04). Можно внести коэффициент металличности K_m и линейно смешивать альбедо и коэффициенты F_0 следующим образом:

```
float3 F0 = F0_0 * (1.0 - Km) + albedo_0 * Km;
float3 albedo = albedo_0 * (1.0 - Km) + F0_0 * Km;
```

При этом цвет металлов задается в параметре albedo_0 и при коэффициенте K_m = 1. Для диэлектриков K_m = 0. В принципе, можно выбирать и другие значения коэффициента K_m, определяя таким образом «металличность» поверхности. Обычно этот коэффициент задают в виде дополнительной текстуры, что позволяет получать различные оттенки металлической поверхности у объекта. На рис. 3.10 представлены варианты поверхности с коэффициентами K_m (0,0; 0,5; 1,0).

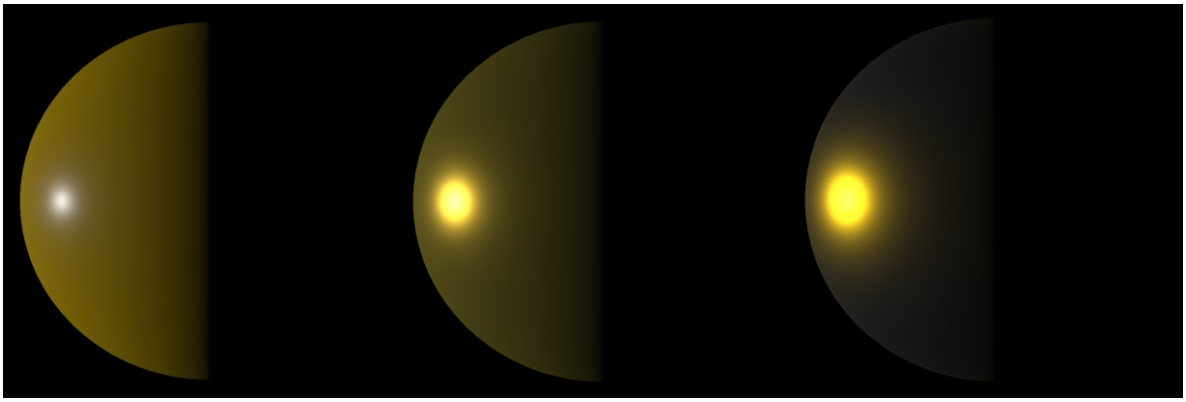


Рис. 3.10. Различные коэффициенты металличности

Остается добавить в шейдер источник рассеянного света ambient и добавлять его к результату PBR, как показано на рис. 3.11. Полные тексты вершинного и пиксельного шейдеров приведены в прил. 3. Пример можно скачать по ссылке: https://github.com/samoyow1973/Example9.

Рис. 3.11. Пример PBR на основе модели Кука – Торренса

Параметры металличности и шероховатости можно передавать в виде текстур. В коде программы устанавливать их, например, так:

```
deviceContext->PSSetShaderResources(1, 1, &roughness);
deviceContext->PSSetShaderResources(2, 1, &metallic);
```

А в пиксельном шейдере определять следующими строками:

```
Texture2D troughness: register(t1);
Texture2D tmetallic: register(t2);
SamplerState SampleType: register(s0);
float Km = tmetallic.Sample(SampleType, input.tex).r;
float roughness = troughness.Sample(SampleType, input.tex).r;
```

Текстуры шероховатости и металличности установлены в текстурные слоты 1 и 2 соответственно. Можно использовать и одну текстуру, например, формата DXGI_FORMAT_R8G8_UINT, устанавливать ее в первый текстурный слот, и тогда в канале *r* будет коэффициент металличности, а в канале *g* – коэффициент шероховатости

```
float Km = troughness.Sample(SampleType, input.tex).r;
float roughness = troughness.Sample(SampleType, input.tex).g;
```

Пример передачи характеристик материала объекта с помощью текстур приведен на рис. 3.12.

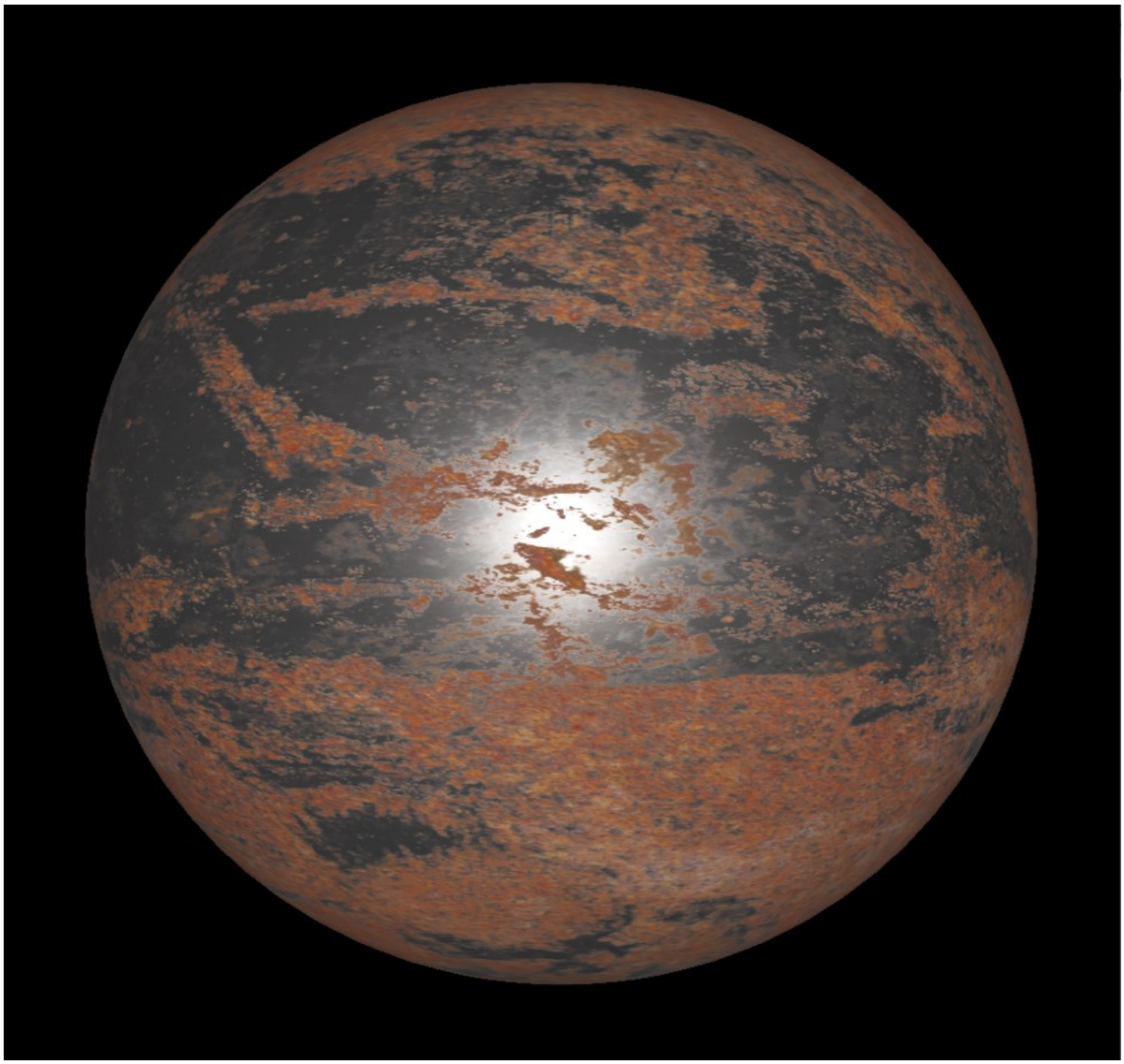


Рис. 3.12. Передача характеристик материала с помощью текстур

3.5. Затухание света

Как известно, энергия света уменьшается пропорционально квадрату расстояния. Поэтому при разработке модели освещения необходимо вносить множитель ослабления, который зависит от расстояния до источника света. Вышесказанное относится только к точечным источникам света. Подразумевается, что направленные источники света, в которых все лучи параллельны, например Солнце,

находятся так далеко от наблюдателя, что в масштабах 3D-сцены нет необходимости учитывать ослабление энергии света.

Наиболее популярный (и так было в ранних версиях DirectX) коэффициент затухания вычисляют по формуле

$$K_{att} = ax^2 + bx + c,$$

где K_{att} – коэффициент затухания; x – расстояние от источника света; a, b, c – коэффициенты, определяющие параметры затухания. В целом эта формула не совсем корректна, но с точки зрения визуализации источников света достаточно удобна. Однако к ее недостаткам относятся необходимость хранить три параметра затухания и отсутствие информации о радиусе действия источника света, которая может потребоваться в дальнейшем.

Некоторые разработчики трехмерной графики используют другую формулу расчета коэффициента ослабления, которую можно выразить на HLSL-коде следующим образом:

```
float Attetuation(float l_min, float l_max, float distance)
{
 float d = max(distance, l_min);
 return saturate(1.0 - pow(d / l_max, 4.0)) / (d * d + 1.0);
}
```

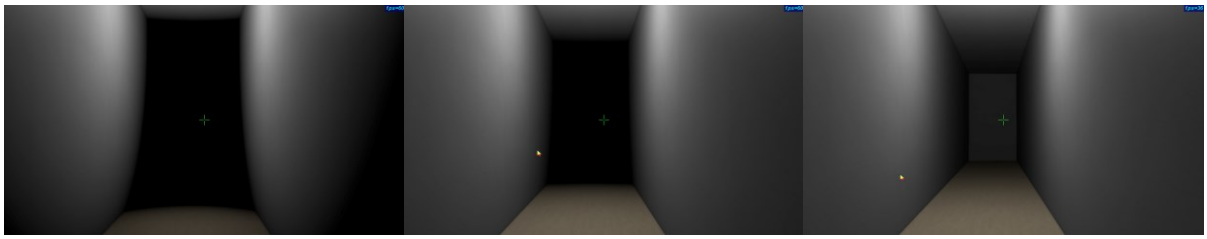
Этот метод основан на двух расстояниях от источника: l_min и l_max. Первый параметр l_min – это минимальное расстояние, меньше которого энергия света не меняется и коэффициент затухания близок к единице (фактически это размер источника света). Второй параметр l_max – это максимальное расстояние, начиная с которого можно не учитывать влияние источника света, т. е. его световая энергия равна нулю (одновременно этот параметр является радиусом действия точечного источника света). Оба этих параметра могут передаваться в шейдер в константном буфере вместе с другими характеристиками источника освещения. Приведенная модель затухания имеет высокую степень адекватности и достаточно точно моделирует реальное затухание энергии света с увеличением расстояния от источника.

Для определения коэффициента затухания необходимо вычислить расстояние, на котором находится источник света,

```
float light_length = length(input.light_pos-w_pos);
float Katt = Attetuation(l_min, l_max, light_length);
```

Но чтобы вычислить расстояние, необходимо знать трехмерные координаты пиксела в мировом пространстве. В пиксельном шейдере такой информации нет, поэтому существует два варианта. В первом варианте необходимо передавать из вершинного шейдера координаты вершин в мировом пространстве, которые будут интерполированы и получены в пиксельном шейдере; а во втором варианте необходимо восстановить позицию текущего пиксела из той информации, которая поступает во входном параметре float4 position : SV_POSITION. Для восстановления потребуются размеры окна в пикселах и обратная матрица произведений матриц перспективы и вида. Функция восстановления позиции приведена в прил. 4.

На рис. 3.13, *а, б, в* приведены примеры освещения коридора для разных характеристик затухания источника света.



а) *б)* *в)*

Рис. 3.13. Примеры разных параметров затухания источника света

3.6. Материалы

Выше были изложены основные принципы освещения трехмерных сцен и приведен ряд примеров. В каждом примере использовались такие характеристики трехмерных объектов, как диффузная составляющая или альбедо, спекулярная составляющая или коэффициенты Френеля F_0, коэффициенты металличности и шероховатости.

Набор этих параметров и образует материал объекта. К материалу можно добавить еще такую составляющую, как свет, который испускает сам материал (т. е. материал светится в темноте); цвет материала для рассеянного света (ambient) и другие характеристики, например, настройки затенения окружением, о чем будет рассказано в п. 4.3. Материал может быть передан в шейдер в виде константного буфера.

Но поскольку передавать параметры материала в константный буфер для каждого примитивного объекта слишком накладно с точки зрения быстродействия, целесообразно свести все используемые материалы в один константный буфер и передавать в шейдер только индекс материала из этого буфера. Причем передавать не для каждой вершины, а для объекта в целом.

Максимальный размер константного буфера 64 Кб. Таким образом, материал содержит 20 переменных типа float, т. е. занимает в памяти 80 байт, то в таком константном буфере поместится 819 разных материалов, что вполне достаточно для большинства задач.

На рис. 3.14 приведен пример вывода на экран одинаковых объектов с разными материалами. По оси *x* увеличивается коэффициент металличности от 0 до 1, по оси *y* увеличивается коэффициент шероховатости от 0,2 до 0,6.

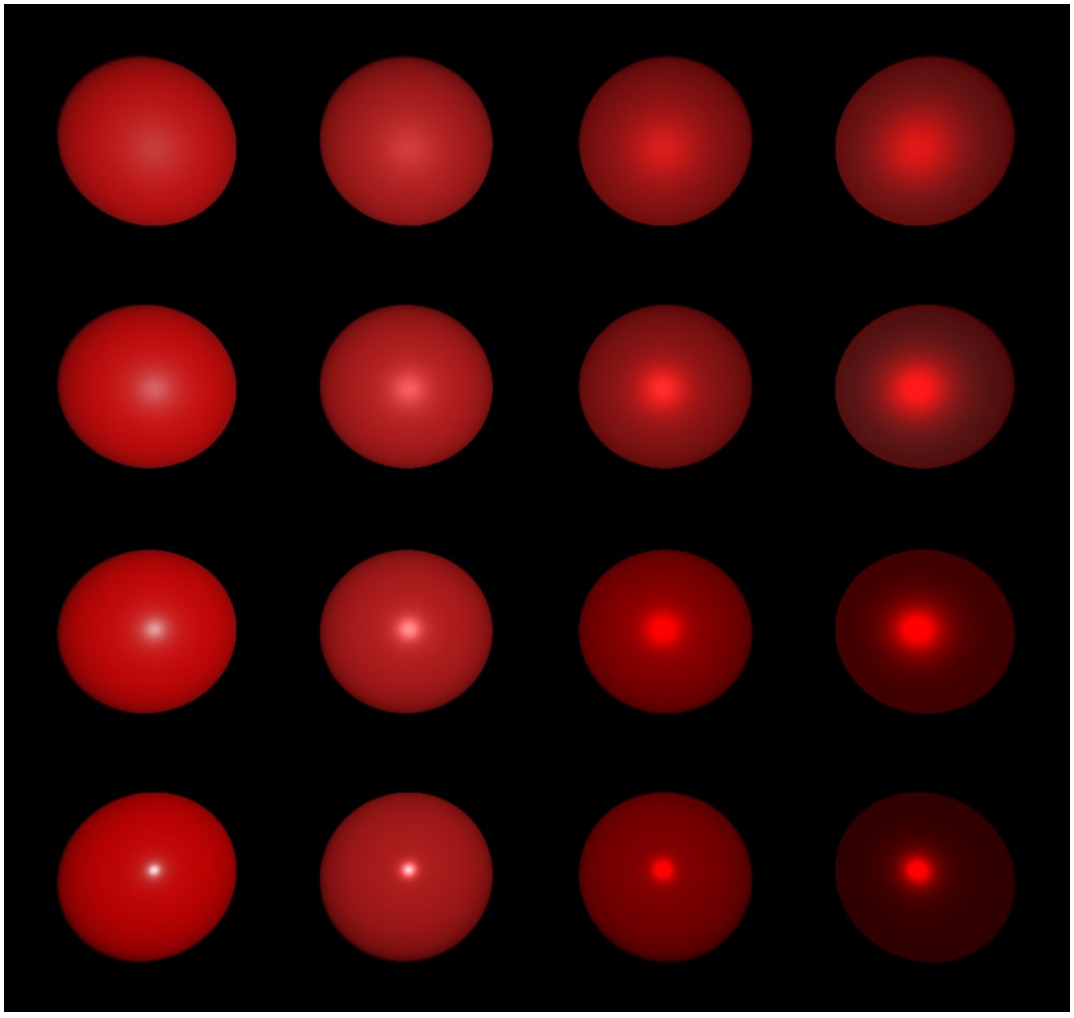


Рис. 3.14. Различные параметры материалов

3.7. Карты окружения

Для придания большей реалистичности в освещении различных объектов с гладкой поверхностью используют метод карт окружения, или Environment mapping. Идея заключается в том, чтобы окружающее пространство отражалось в объектах с низкой степенью шероховатости. Для этого используют специальные текстуры, которые представляют собой окружающее пространство. Например, это может быть кубическая текстура, как показано на рис. 3.15. Она состоит из шести текстур, которые являются стенками куба.

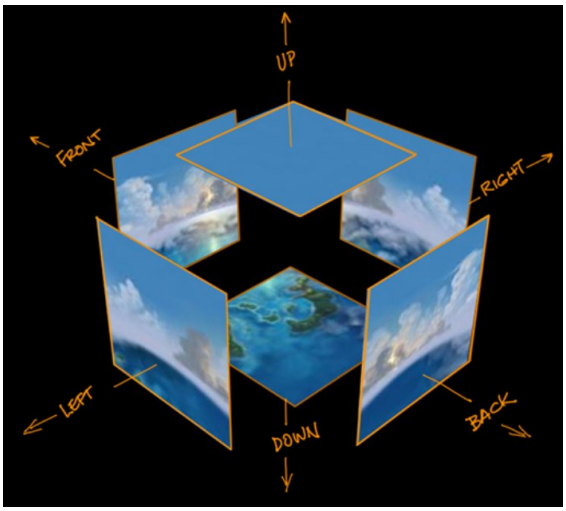


Рис. 3.15. Кубическая текстура

На плоскости эта текстура выглядит так, как показано на рис. 3.16. Нетрудно заметить, что такой вариант развертки легко складывается в куб.

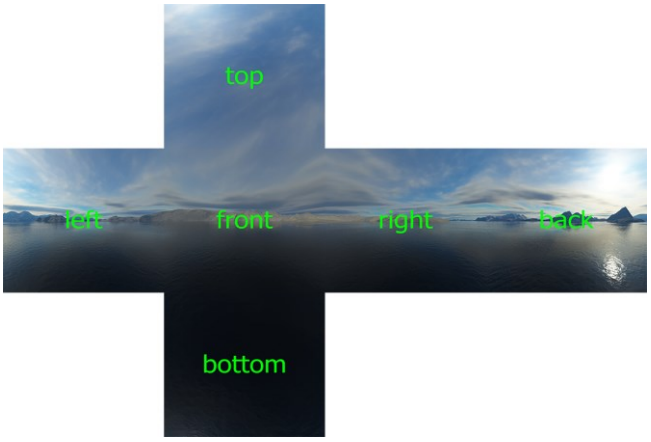


Рис. 3.16. Развертка кубической текстуры

Поскольку объект находится в центре такого куба, необходимо вычислить вектор отражения взгляда, чтобы определить, что из этой текстуры переносить на объект. Рассмотрим реализацию этого метода подробнее.

Сначала необходимо загрузить кубическую текстуру. Загрузка кубической текстуры в память аналогична загрузке обычной текстуры. Далее, так же как и обычную текстуру, необходимо установить кубическую текстуру в один из текстурных регистров, чтобы иметь к ней доступ из пиксельного шейдера.

В пиксельном шейдере описываем эту текстуру как кубическую

TextureCube Texture: register(t0);

SamplerState Sample0: register(s0);

Теперь необходимо рассчитать вектор отражения. Вектор отражения рассчитывают между вектором взгляда и нормалью

float3 ref = reflect(-V, N);

Вектор взгляда возьмем отрицательным, потому что в этом случае он должен указывать на объект, а не на позицию камеры. Остается считать значение из кубической текстуры

float3 r_albedo = Texture.Sample(Sample0, ref).rgb;

Теперь, перемножив альбедо материала и альбедо текстуры отражения, получим различные материалы с отражением, как показано на рис. 3.17, для разных коэффициентов шероховатости – от 0,3 до 0,6.

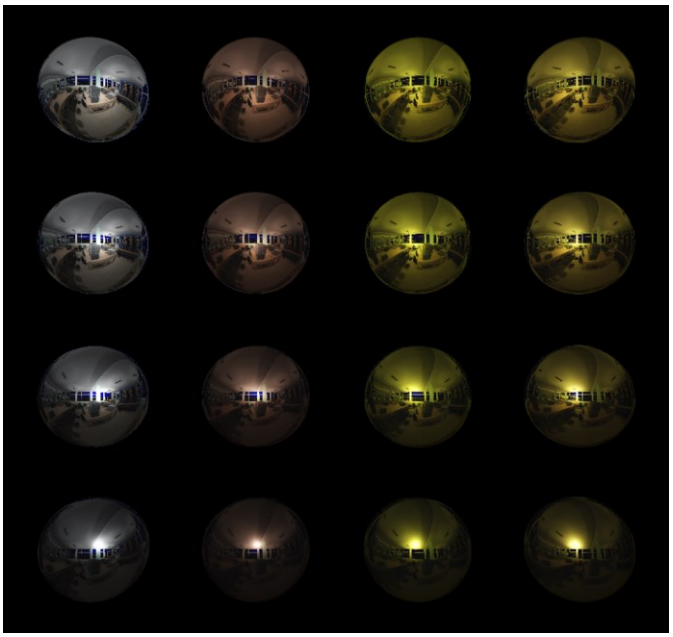


Рис. 3.17. Применение карты окружения

Конечно, такой метод зеркальных отражений еще далек от реальности, поскольку качество отражения должно зависеть от шероховатости материала, и кроме этого необходимо учитывать непрямое диффузное освещение, но изучение данного вопроса выходит за рамки настоящего учебного пособия. Тем не менее даже упрощенное решение вопроса отражений позволяет получить более реалистичную картинку, как видно на рис. 3.18, *а* (без отражений) и 3.18, *б* (с отражениями в кубках и бокалах).

а) *б)*

Рис. 3.18. Сравнение отображений при использовании environment mapping

3.8. Карты нормалей

Метод использования карт нормалей (normal mapping, или bump mapping) применяют для имитации неровностей на поверхности объекта. Дело в том, что освещение поверхности в первую очередь зависит от ее нормали в каждой точке. Ранее мы рассматривали только такие случаи, когда нормаль всех точек поверхности была одинаковая либо линейно менялась. Карты нормалей позволяют в каждой точке поверхности устанавливать свою, индивидуальную нормаль, тем са-

мым делая уровень ее освещения отличным от соседних точек и фактически добавляя некоторые неровности к поверхности. На рис. 3.19 показано, как изменение нормалей отдельных фрагментов позволяет имитировать неровности.

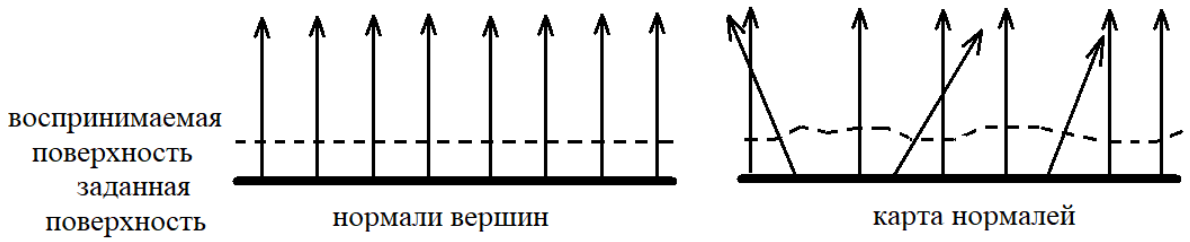
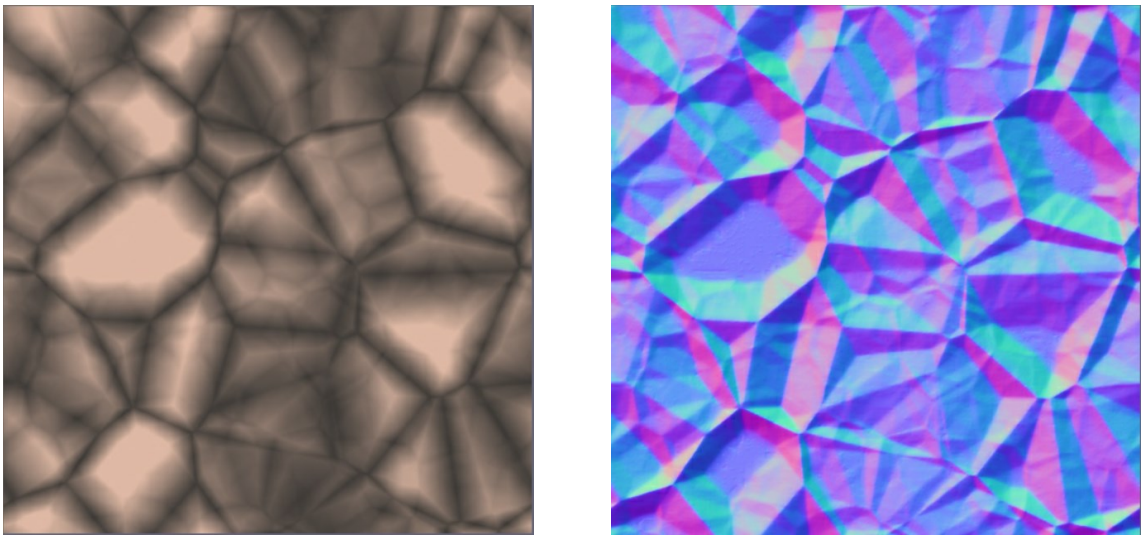


Рис. 3.19. Имитация неровностей картой нормалей

Алгоритм освещения, таким образом, начнет считать поверхность объекта множеством мелких плоскостей и освещать их в соответствии уже с их нормалями. Это позволит имитировать углубления, трещины и различные неровности.

Карта нормалей обычно хранится в текстуре в формате RGB. Только вместо цветовых компонент там хранятся компоненты x, y, z нормали в каждой точке объекта. При этом следует отметить, что нормализованные векторы имеют диапазон своих компонент от –1 до 1, а цветовые компоненты способны хранить диапазон от 0 до 1. Поэтому вектор нормали для хранения в текстуре вычисляют по формуле $n' = n/2 + 0,5$.

На рис. 3.20, *а* представлена текстура камней, а на рис. 3.20, *б* – карта нормалей для этой текстуры. Карты нормалей преимущественно имеют синеватый оттенок. Связано это с тем, что компонента z нормали записывается в синий канал, а направление нормали, как правило, в основном направлено к нам. Однако это вызывает определенные проблемы. Дело в том, что заданный программистом треугольник может быть произвольно ориентирован в пространстве, а карта нормалей останется прежней. Таким образом, в пиксельном шейдере мало считать нормаль из карты нормалей, еще необходимо ее повернуть в нужную сторону.



а) *б)*

Рис. 3.20. Текстура и карта нормалей

Для этого метода используют пространство TBN (tangent, binormal, normal), называемое также *касательным пространством*. По сути, эти три вектора: tangent, binormal, normal – представляют собой единичные векторы обычного трехмерного пространства, перпендикулярные между собой.

Для применения карты нормалей каждая вершина кроме своих атрибутов должна будет иметь в обязательном порядке нормаль, бинормаль и тангенту. В прил. 5 приведена программа расчета трех векторов в пространстве TBN для трех вершин треугольника. Расчеты производятся для каждой вершины трехмерного объекта. Для этого необходимо в цикле пройти по всем треугольникам объекта, рассчитать для его вершин векторы в пространстве TBN и записать их в атрибуты каждой вершины.

Структура вершины может быть такой:

```
struct VERTEX
{
        XMFLOAT3 pos;
        XMFLOAT3 normal;
        XMFLOAT3 tangnt;
        XMFLOAT3 binormal;
        XMFLOAT4 color;
        XMFLOAT2 tex;
};
```

Потребуется и изменение описания входных данных для вершинного шейдера, это может выглядеть, например, так:

```
desc[2].SemanticName = "TANGENT";
desc[2].SemanticIndex = 0;
desc[2].Format = DXGI_FORMAT_R32G32B32_FLOAT;
desc[2].InputSlot = 0;
desc[2].AlignedByteOffset =
D3D11_APPEND_ALIGNED_ELEMENT;
desc[2].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
desc[2].InstanceDataStepRate = 0;
desc[3].SemanticName = "BINORMAL";
desc[3].SemanticIndex = 0;
desc[3].Format = DXGI_FORMAT_R32G32B32_FLOAT;
desc[3].InputSlot = 0;
desc[3].AlignedByteOffset =
D3D11_APPEND_ALIGNED_ELEMENT;
desc[3].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
desc[3].InstanceDataStepRate = 0;
```

В вершинном шейдере необходимо изменить описание входных и выходных данных, например:

```
struct INPUT
{
 float3 position : POSITION;
 float3 normal : NORMAL;
 float3 tangent: TANGENT;
 float3 binormal: BINORMAL;
 float4 color : COLOR0;
 float2 tex : TEXCOORD0;
};
struct OUTPUT
{ float4 position : SV_POSITION;
 float3 normal : NORMAL;
 float3 tangent: TANGENT;
 float3 binormal: BINORMAL;
 float2 tex:TEXCOORD0;
 float3 V:TEXCOORD1;
 float3 L:TEXCOORD2;
 float3 light_pos:TEXCOORD3;
 float3 w_pos:TEXCOORD4;}
```

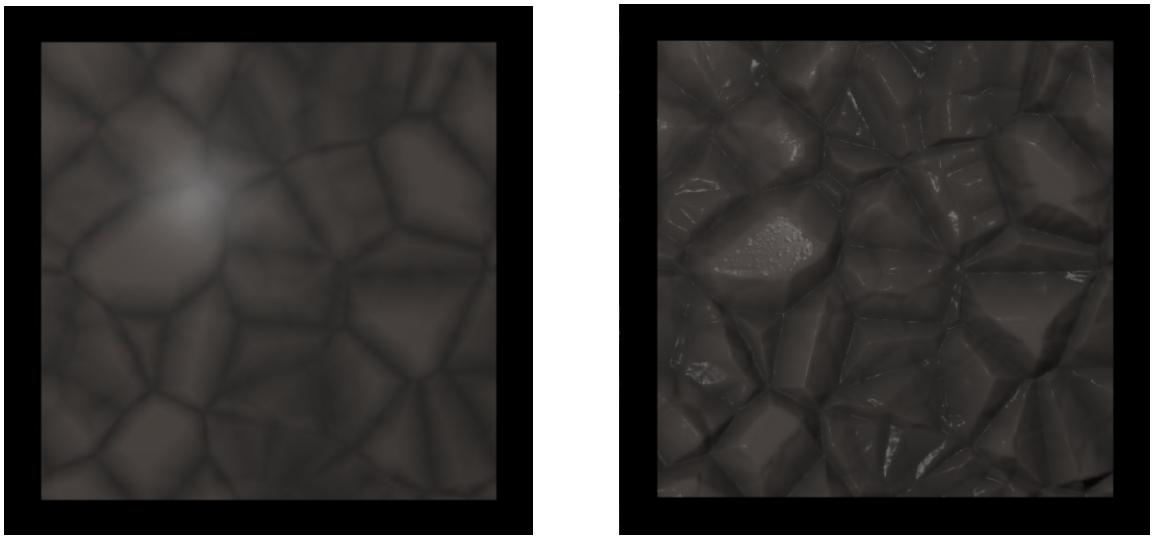
Необходимо еще не забыть передать тангенту и бинормаль в пиксельный шейдер

```
output.tangent = input.tangent;
output.binormal = input.binormal;
```

Структура входных данных пиксельного шейдера совпадает со структурой выходных данных вершинного шейдера. Преобразования значений из карты нормалей осуществляются следующим образом:

```
float3 bumpMap;
float3 bumpNormal;
input.normal = normalize(input.normal);
input.binormal = normalize(input.binormal);
input.tangent = normalize(input.tangent);
// Считать нормаль из карты нормалей.
bumpMap = NormalTexture.Sample(SampleType, input.tuv).xyz;
// Преобразовать диапазон значений из (0, +1) в (-1, +1).
bumpMap = (bumpMap * 2.0f) - 1.0f;
// Рассчитать нормаль из карты нормалей.
bumpNormal = (bumpMap.x * input.tangent) + (bumpMap.y * input.binormal) + (bumpMap.z * normal);
// Нормализовать результат
bumpNormal = normalize(bumpNormal);
```

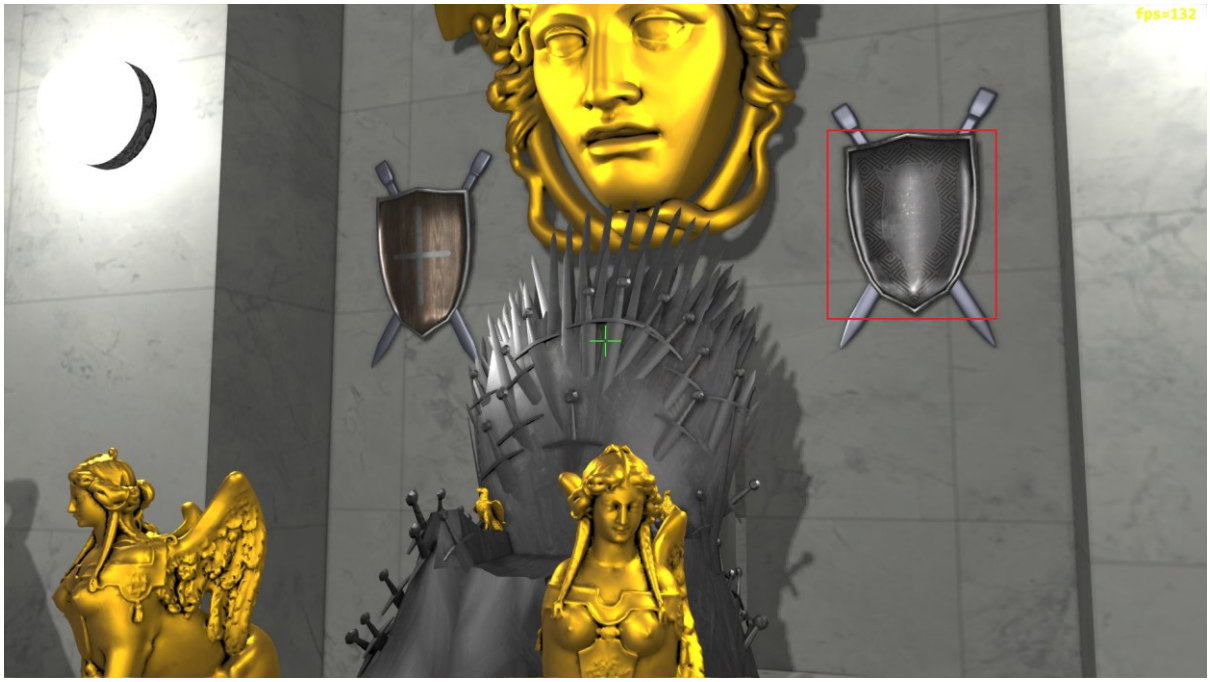
На рис. 3.21, *а*, *б* показан результат без применения карты нормалей и с ней.



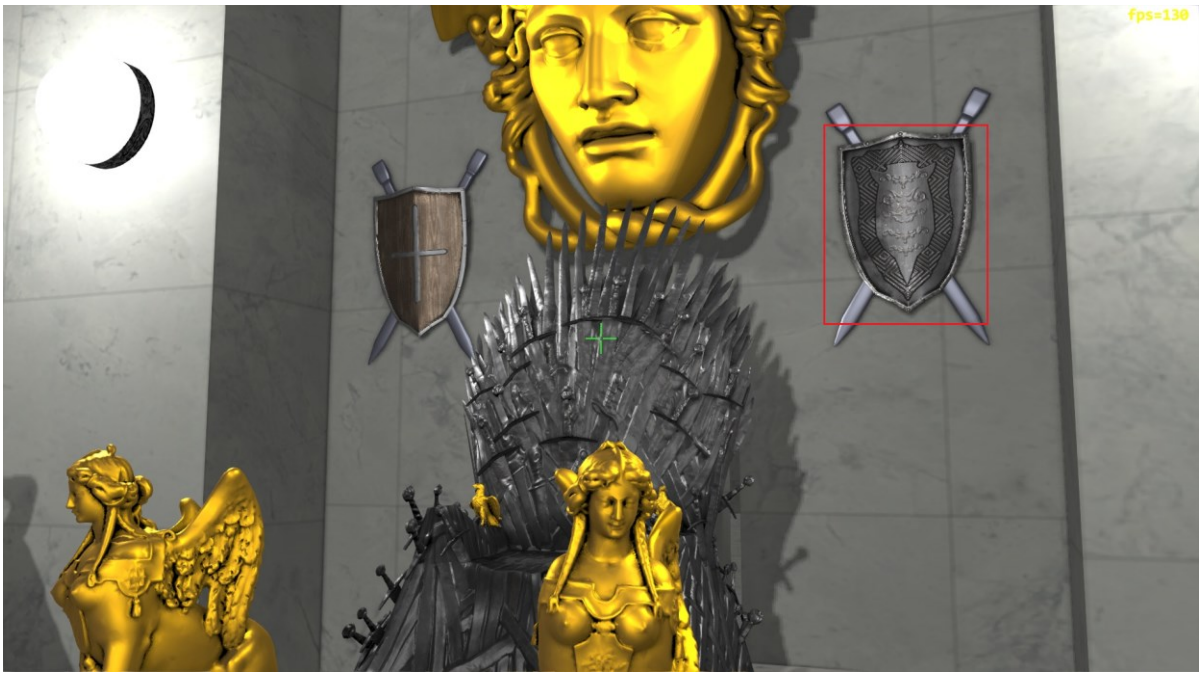
а) *б)*

Рис. 3.21. Эффект применения карт нормалей

На рис. 3.22 показаны возможности метода карт нормалей, позволяющие существенно увеличить детализацию объекта без увеличения количества треугольников, из которых он состоит.



а)



б)

Рис. 3.22:
а – 3D-сцена с отключенными картами нормалей;
б – 3D-сцена с картами нормалей

Детализация щита в правом верхнем углу рис. 3.22 выполнена практически только за счет карты нормалей, изображенной на рис. 3.23. Пример использования карт нормалей можно скачать по ссылке: https://github.com/samoyow1973/Example10.

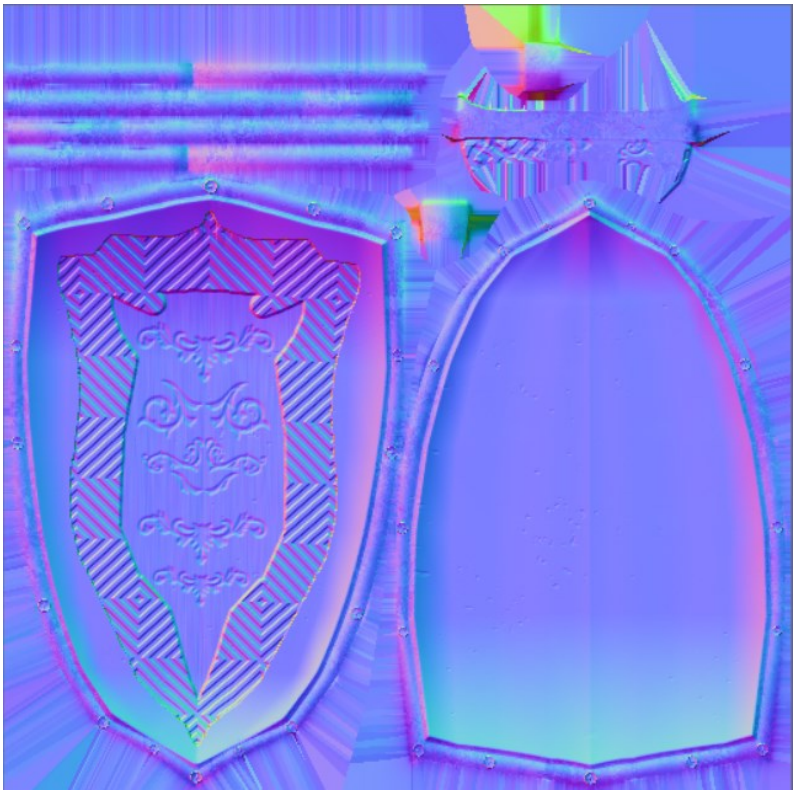


Рис. 3.23. Карта нормалей для детализации объекта

Контрольные вопросы

1. Какие типы источников света используют в компьютерной графике?
2. Какие виды света используют в моделях освещения?
3. Каковы самые известные модели освещения?
4. В чем особенности затенения по Гуро?
5. В чем достоинства и недостатки модели затенения по Гуро?
6. В чем отличие модели затенения по Фонгу от модели затенения по Гуро?
7. В чем заключается оптимизация Блинна в модели затенения по Фонгу?
8. Что такое гамма-коррекция?
9. В чем заключается основной недостаток моделей затенения по Гуро и по Фонгу?

10. Какими свойствами должна обладать физически корректная модель затенения?
11. В чем особенности модели затенения Кука – Торренса?
12. В чем заключается суть метода Смита?
13. Что означает понятие «двухлучевая модель затенения»?
14. Что такое коэффициент шероховатости поверхности?
15. Что такое коэффициенты Френеля?
16. В чем отличие металлов от диэлектриков в модели Кука – Торренса?
17. В чем заключается суть аппроксимации Шлика?
18. Что такое альбедо?
19. Как передавать параметры шероховатости и металличности с помощью текстур?
20. Приведите пример модели затухания света в зависимости от расстояния от источника освещения.
21. Каковы основные параметры материалов?
22. В чем заключается суть метода карт окружения?
23. Что такое кубическая текстура?
24. В чем заключается суть метода карт нормалей?
25. Что такое касательное пространство TBN?
26. Как рассчитать бинормаль и тангенту?

Глава 4. ОТЛОЖЕННОЕ ОСВЕЩЕНИЕ

4.1. G-буфер

В предыдущей главе были рассмотрены некоторые варианты реализации затенения и освещения трехмерных объектов. Практически все эти варианты требуют каких-либо расчетов в пиксельном шейдере, зачастую достаточно ресурсоемких. Причем расчеты эти должны быть произведены для всех источников света, которые присутствуют в 3D-сцене. Таким образом, при выводе объекта необходимо кроме его характеристик (материалов и текстур) передать в шейдеры список источников света и просчитать освещение объекта каждым из них. Такой подход называют *прямым освещением,* или *затенением* (forward shading), или *прямым рендерингом* (forward rendering).

Следует отметить, что данный вариант не лишен ряда недостатков. Во-первых, он ограничивает количество источников света в сцене, потому что в шейдерах придется рассматривать их все, для всех выводимых объектов. Циклы по *N* источникам света резко снизят количество выводимых кадров в секунду fps (frames per second). Во-вторых, множественные и ресурсоемкие расчеты могут оказаться напрасными, если объект, для которого они производились, впоследствии будет полностью или частично загорожен другим объектом. Действительно, рассчитывать освещение пиксела, когда поверх него в этом же кадре может быть выведен другой пиксел, – бесполезная трата вычислительных ресурсов.

Альтернатива прямому освещению и затенению – *метод отложенного освещения или затенения* (deferred shading), который заключается в том, что первоначально трехмерная сцена выводится не в задний буфер (back buffer), а в специальный G-буфер без какого-либо расчета освещения. В G-буфер заносятся все необходимые компоненты для последующего расчета. Это альбедо, спекулярная составляющая (коэффициенты Френеля), нормали, индекс материала объекта и т. п. Другими словами, в G-буфер помещается информация, которая имеет отношение к объекту и не относится к источнику света. Рассмотренные ранее методы карт окружения и карт нормалей тоже используются на этом этапе.

Информация о положении пиксела в пространстве может быть получена из координат пиксела на экране и из значения в буфере глубины, как это было показано в п. 3.5 и прил. 4. Именно поэтому в п. 2.1 при инициализации DirectX буфер глубины сразу был создан не только для своих прямых целей, но и как ресурс шейдера, чтобы впоследствии его можно было подключить как текстуру и получать из него данные

device->CreateShaderResourceView(pointer,&descSRV,&depthStencilTexture)

Второй этап – этап расчета затенения и освещения для каждого источника света. Но с учетом того, что точечные источники света имеют ограниченный радиус действия, расчет будет производиться не для всех пикселов в G-буфере, а только для тех, которые могут быть потенциально освещены. Такой подход дает возможность использовать в трехмерной сцене достаточно большое количество источников света. Подробнее об алгоритме расчета освещения при отложенном рендеринге будет изложено в следующих параграфах.

Тем не менее метод отложенного освещения хоть и может дать выигрыш в производительности, но тоже не лишен недостатков. Во-первых, это критичность к размеру G-буфера. Чем меньше G-буфер и разрядность его ячеек (пикселов), тем выше производительность. Поэтому тщательное планирование составляющих G-буфера – залог высокого показателя fps при отображении трехмерных сцен в режиме реального времени. Во-вторых, отложенное освещение лишено возможности вывода прозрачных объектов, потому что невозможно хранить информацию в одной ячейке (пикселе) G-буфера о двух или более объектах сразу. Вопрос отображения прозрачных объектов при отложенном освещении будет описан в соответствующем параграфе.

Таким образом, G-буфер представляет собой несколько поверхностей рендеринга, одновременно являющихся ресурсами шейдера, т. е. впоследствии их можно использовать как текстуры. Рассмотрим подробнее создание таких поверхностей.

Сначала надо заполнить структуру D3D11_TEXTURE2D_DESC необходимыми параметрами

D3D11_TEXTURE2D_DESC desc;

desc.ArraySize = 1; // количество текстур в массиве

// Флаги, показывающие, что наша текстура предназначена для использования в шейдерах и для рендеринга,

```
desc.BindFlags = D3D11_BIND_SHADER_RESOURCE |
D3D11_BIND_RENDER_TARGET;
desc.CPUAccessFlags = 0; // доступ из CPU к текстуре
desc.Format = DXGI_FORMAT_R8G8B8A8_UNORM; // формат текстуры
desc.Height = windowHeight; // размер текструры
desc.MipLevels = 1; // количество mip уровней
desc.MiscFlags = 0; // дополнительные флаги
desc.SampleDesc.Count = 1; // Параметры для мультисэмплов
desc.SampleDesc.Quality = 0; // Параметры для мультисэмплов
desc.Width = windowWidth; // размер текстуры
desc.Usage = D3D11_USAGE_DEFAULT; // параметры записи/чтения
```

Далее необходимо создать текстуру в памяти, получить указатель на нее и создать как ресурс для шейдера, так и поверхность рендеринга

```
ID3D11Texture2D* pointer=NULL;
ID3D11RenderTargetView* target=NULL;
ID3D11ShaderResourceView* texture=NULL;
if (FAILED(device->CreateTexture2D(&desc, NULL, &pointer)))
return E_FAIL;
if (FAILED(device->CreateShaderResourceView(pointer, NULL, &texture)))
                                                    return E_FAIL;
if (FAILED(device->CreateRenderTargetView(pointer, NULL, &target)))
                                                    return E_FAIL;
ReleaseCOM(pointer);
```

Необходимо вдумчиво подходить к выбору формата поверхности. Дело в том, что компоненты нормали могут требовать при хранении большей точности, чем компоненты альбедо. Поэтому для хранения нормалей лучше создавать текстуру с форматом DXGI_FORMAT_R16G16B16A16_FLOAT, обладающим большей точностью. С другой стороны, чем меньше размер G-буфера в целом

и чем меньше размер его поверхностей для хранения информации, тем выше быстродействие. Установить поверхности вывода и буфер глубины возможно следующей функцией:

deviceContext->OMSetRenderTargets(count, targets, depth_stencil), где count – количество поверхностей рендеринга; targets – имя массива с поверхностями рендеринга; depth_stencil – указатель на буфер глубины.

Теперь из пиксельного шейдера можно выводить информацию на любую поверхность рендеринга. Для этого в пиксельном шейдере необходимо создать структуру выходных данных, например, такую:

```
struct PS_OUTPUT
{
        vector diffuse  : SV_TARGET0;
        vector normal  : SV_TARGET1;
        vector specular  : SV_TARGET2;
};
```

где SV_TARGET*N* – поверхность рендеринга с номером *N*.

Выходные данные пиксельного шейдера формируются как обычно, например:

```
output.diffuse = float4(Texture.Sample(SampleType, input.tuv).rgb, roughness);
output.normal = float4(normal, index_material);
output.specular = float4(specular, metallic);
```

Как можно заметить, организовать первый этап отложенного освещения не так уж и сложно. На рис. 4.1, *а* – *е* представлен пример всех поверхностей, составляющих G-буфер при отображении трехмерной сцены, и информация в виде цвета, которая в этом буфере содержится: альбедо, нормали (в виде цветов), спекулярная составляющая, буфер глубины, материалы и сама трехмерная сцена после применения всех задействованных источников света. Об алгоритме освещения при отложенном затенении и пойдет речь в следующем параграфе.

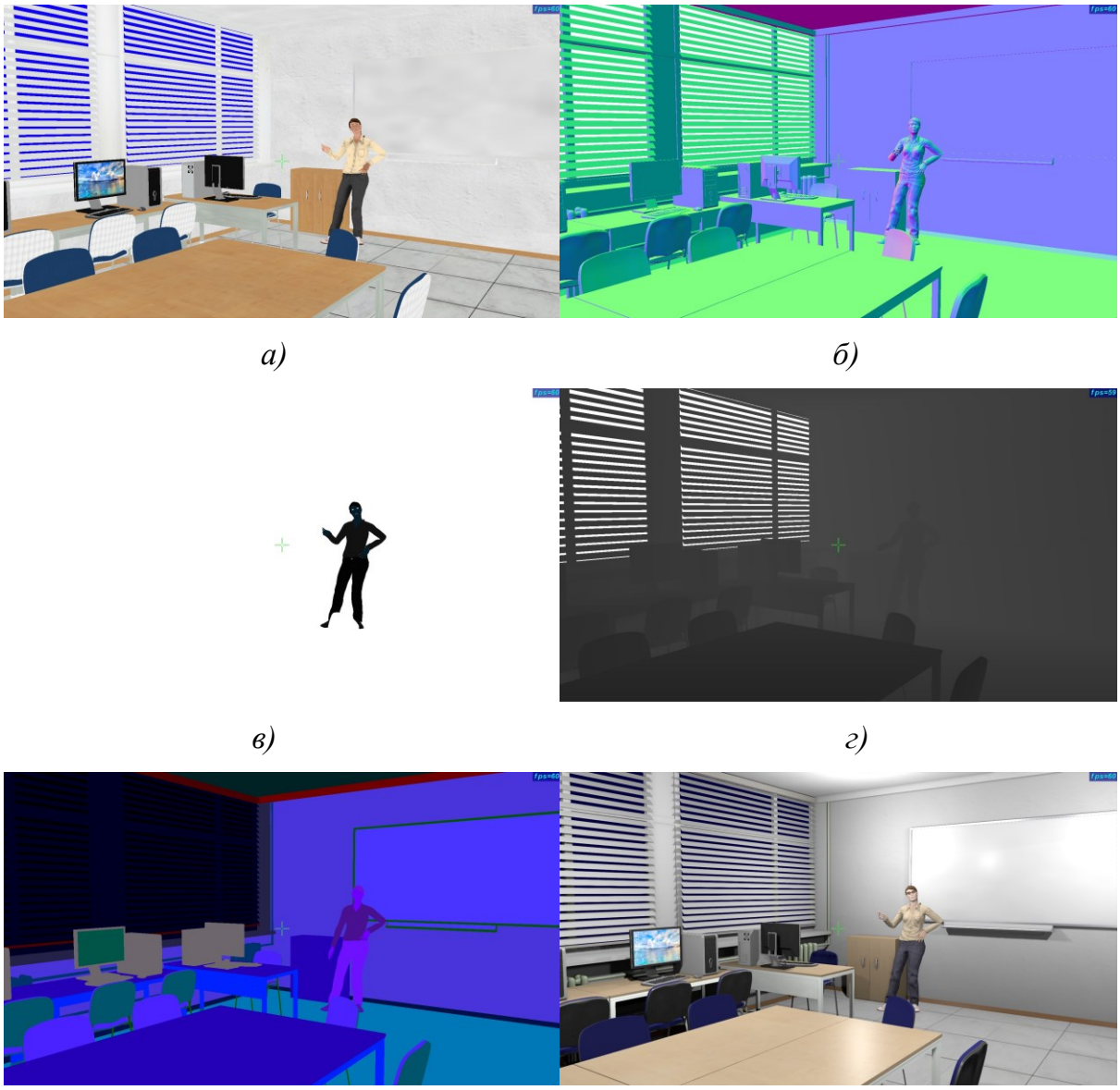


Рис. 4.1. Поверхности G-буфера: а – альбедо; б – нормали; в – спекулярная составляющая; г – буфер глубины; д – материалы; е – 3D-сцена полностью

4.2. Источники света при отложенном освещении

Метод отложенного освещения дает возможность использовать в сцене десятки источников света. Таким образом, можно на основе информации из G-буфера для каждого пиксела многократно проводить расчеты освещения для различных источников света. Результаты расчетов для каждого пиксела суммируются по всем источникам света, участвующим в сцене. Реализовать это возможно, установив для цели рендеринга финальную поверхность (например, задний буфер) и

выбрав нужное состояние смешивания. Финальную поверхность легко установить уже рассмотренной ранее функцией

deviceContext->OMSetRenderTargets(1, &target, depth_stencil),

где target – указатель на задний буфер.

Нужное состояние смешивания определяется структурой D3D11_BLEND_DESC, функцией создания этого состояния смешивания и установки этого состояния перед началом расчета освещения, как это было показано в п. 2.4. Поскольку в сцене предполагается наличие нескольких источников света, то результаты освещения каждого пиксела должны суммироваться. Именно режим суммирования и необходимо выставить в состоянии смешивания. Для этого поля структуры D3D11_BLEND_DESC заполняют следующим образом:

```
descb.AlphaToCoverageEnable = FALSE;
descb.IndependentBlendEnable = FALSE;
descb.RenderTarget[0].BlendEnable = TRUE; // включить смешивание
descb.RenderTarget[0].SrcBlend = D3D11_BLEND_ONE; // коэффициент
                                        // умножения на выводимый пиксел
descb.RenderTarget[0].DestBlend =
D3D11_BLEND_ONE;//коэффициент
                                        // умножения на уже выведенный пиксел
descb.RenderTarget[0].BlendOp =
D3D11_BLEND_OP_ADD;//сложение
                                        // выводимого пиксела и уже выведенного
// Установки смешивания альфа-компоненты
// в данном случае не играют роли
descb.RenderTarget[0].SrcBlendAlpha = D3D11_BLEND_ONE;
descb.RenderTarget[0].DestBlendAlpha = D3D11_BLEND_ZERO;
descb.RenderTarget[0].BlendOpAlpha=D3D11_BLEND_OP_ADD;
descb.RenderTarget[0].RenderTargetWriteMask = 15; // выводим все цвета
```

Следует помнить, что перед началом расчета освещения необходимо очистить финальную поверхность рендеринга. Потому что ее пикселы будут уже суммироваться с пикселами, выводимыми при расчете первого источника света.

Теперь следует установить настройки буфера глубины. Необходимо отключить запись в буфер глубины (в поле DepthWriteMask структуры D3D11_DEPTH_STENCIL_DESC записать ноль). Далее создать состояние буфера глубины и установить его, как было показано в п. 2.4. Но необходимо понимать, что отключение записи в буфер глубины не отключает сам буфер. Проверки на глубину выводимого пиксела остаются. И пиксел не будет выведен, если проверки не прошли.

По идее, следует отключить сам *z*-буфер тоже, и для каждого источника света, будь он точечным или направленным, выводить четырехугольник по размеру экрана и в пиксельном шейдере производить расчет освещения каждого пиксела экрана. Для точечных источников можно использовать сравнение на расстояние от источника света, и если это расстояние больше радиуса действия источника, то завершать расчет функцией discard. Но, во-первых, время выполнения шейдера зависит от самого длинного пути его выполнения и discard в большинстве случаев не способен дать какой-то выигрыш. Таким образом, расчет освещения будет «как бы» выполняться для всех пикселов экрана и для всех источников освещения. Во-вторых, проверки на дальность пиксела от источника света тоже занимают время. Поэтому можно предложить ряд оптимизаций.

Первая оптимизация связана с тем, что для точечных источников можно выводить шар с радиусом действия источника света. Таким образом, будут обработаны только те пикселы, которые затронет этот шар. Правда, не все пикселы экрана попадут внутрь шара, он, возможно, их только загородит. И расчет освещения будет произведен в том числе для пикселов, которые находятся очень далеко от источника освещения. Вторая оптимизация связана с использованием буфера трафарета.

Буфер трафарета неразрывно связан с буфером глубины. По сути, это два буфера в одном. При создании буфера глубины в п. 2.1 изначально был выбран формат

DXGI_FORMAT_D24_UNORM_S8_UINT,

т. е. в 24 бита ячейки буфера записывается глубина, а в 8 бит этой же ячейки – значение трафарета. Буфер трафарета – достаточно мощный инструмент при реализации различных алгоритмов трехмерной графики.

В буфер трафарета можно записывать какие-либо значения, даже если выводимый пиксел не прошел тест глубины. Так же и с буфером глубины: необходимо устанавливать функцию сравнения, определяющую, когда отбрасывать выводимый пиксел, а когда рисовать. Все эти действия производятся аппаратно, а значит, с высоким быстродействием. Как устанавливать значение, которое будет использовано в тестах трафарета, и как управлять записью в буфер трафарета, вкратце рассказывалось в п. 2.4. Рассмотрим подробнее возможности буфера трафарета.

Управление буфером трафарета осуществляется установкой необходимых полей в структуре D3D11_DEPTH_STENCIL_DESC, созданием состояния буфера глубины и трафарета и установкой этого состояния перед вызовом функций Draw и DrawIndexed.

Буфер трафарета можно включать и выключать полем StencilEnable. При включенном буфере трафарета имеется возможность обработки лицевых или обратных сторон выводимых треугольников, алгоритмы при этом могут быть разными. Каждый алгоритм задается отдельно за счет заполнения структур, за которые отвечают поля FrontFace и BackFace. В целом алгоритм работы с буфером трафарета строится следующим образом:

– в поле StencilFunc устанавливается функция сравнения со значением value, переданным при установке состояния буфера глубины и буфера трафарета. Эта функция будет определять, пройден тест трафарета или нет;

– если тест трафарета не пройден, то поле StencilFailOp определяет, как изменять значение в буфере трафарета. Можно оставить без изменений, записать ноль, увеличить или уменьшить на единицу с переполнением или без, инвертировать или записать значение value;

– если тест трафарета пройден, а тест глубины не пройден, то поле StencilDepthFailOp определяет, как изменять значение в буфере трафарета;

– если оба теста пройдены, то поле StencilPassOp определяет, как изменять значение в буфере трафарета.

Можно предложить следующий алгоритм расчета освещения для каждого точечного источника. Алгоритм заключается в том, что-

бы в буфере трафарета отметить только те пикселы, которые находятся внутри шара с радиусом, равным действию источника света, и центром, совпадающим с его позицией. Алгоритм состоит из трех пунктов, в которых необходимо произвести ряд настроек состояний смешивания, растеризации и буфера глубины и трафарета.

1. Отключить возможность вывода на поверхность рендеринга (значение поля RenderTargetWriteMask структуры состояния смешивания установить в ноль), а смешивание отключить (поле BlendEnable=0). Настроить состояние буфера глубины и трафарета так, чтобы отключить запись в буфер глубины (поле DepthWriteMask=0), но оставить его включенным для проверок. Включить буфер трафарета (поле StencilEnable). Заполнить поля структуры BackFace

```
BackFace.StencilFailOp = D3D11_STENCIL_OP_KEEP;
BackFace.StencilDepthFailOp = D3D11_STENCIL_OP_INCR;
BackFace.StencilPassOp = D3D11_STENCIL_OP_KEEP;
BackFace.StencilFunc = D3D11_COMPARISON_EQUAL;
```

При установке состояния буфера глубины и трафарета значение value установить нулевым. Установить состояние растеризации только для «обратных» треугольников (поле CullMode структуры D3D11_RASTERIZER_DESC). Пиксельный шейдер установить в NULL, поскольку никакой обработки пикселов не требуется. Вывести «световой шар», т. е. трехмерный объект в виде шара с радиусом, совпадающим с радиусом действия источника света, и центром, совпадающим с позицией источника света.

Таким образом, на поверхность отображения ничего выведено не будет. Единственное, что изменится, – буфер трафарета увеличит свое значение на единицу в тех точках, которые ближе к наблюдателю, чем поверхность выводимого шара, т. е. в тех точках, где не прошел тест буфера глубины. Пикселы, где буфер трафарета равен единице, это пикселы, которые потенциально могут быть освещены. На рис. 4.2 показан первый этап алгоритма. Ось Z – это направление, перпендикулярное плоскости отображения. Красным цветом указаны пикселы объектов, для которых значение в буфере трафарета будет установлено в единицу.

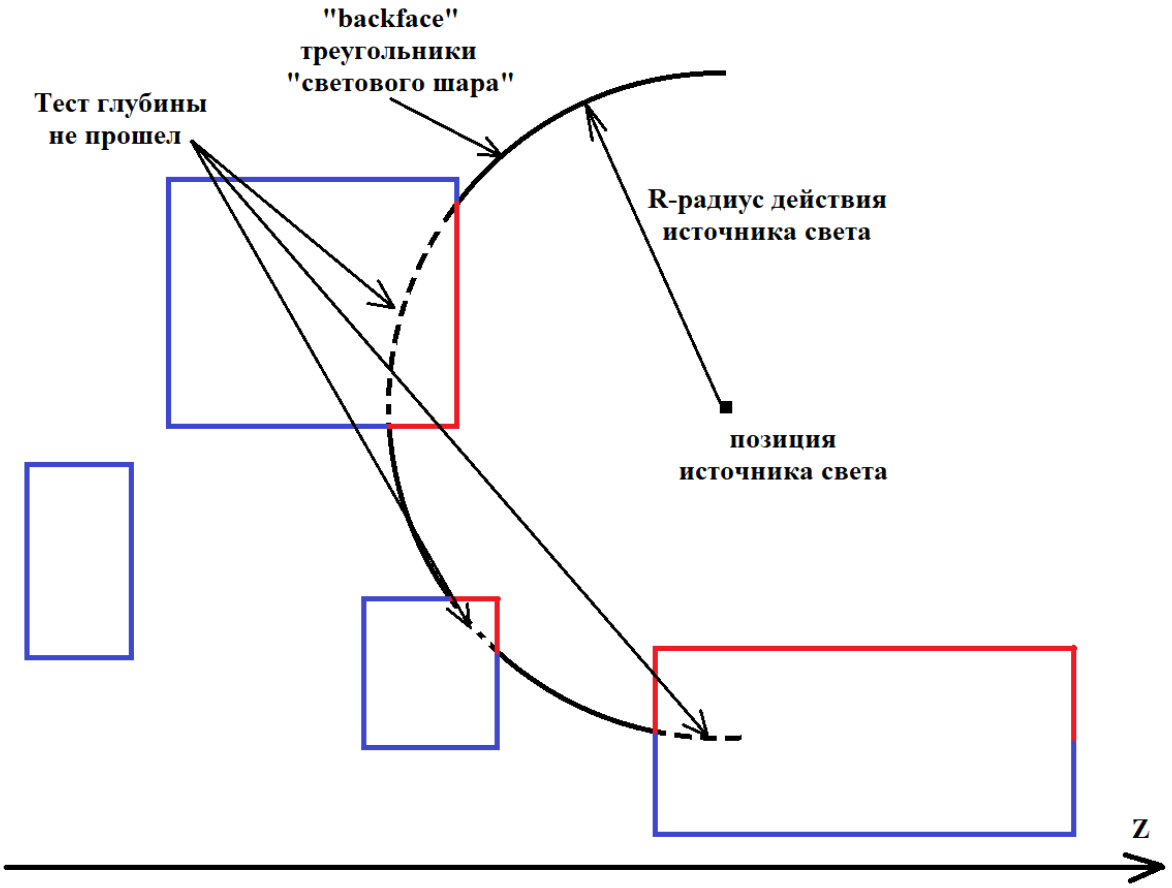


Рис. 4.2. Первый этап алгоритма освещения

2. Установить состояние растеризации только для «лицевых» треугольников (поле CullMode структуры D3D11_RASTERIZER_DESC).

Заполнить поля структуры FrontFace:

FrontFace.StencilFailOp = D3D11_STENCIL_OP_KEEP;

FrontFace.StencilDepthFailOp = D3D11_STENCIL_OP_DECR;

FrontFace.StencilPassOp = D3D11_STENCIL_OP_KEEP;

FrontFace.StencilFunc = D3D11_COMPARISON_EQUAL;

Значение value установить равным единице. На этом шаге будут обработаны только те пикселы, где значение трафарета равно единице, т. е. они могут входить в сферу действия источника света. Но если тест буфера глубины не пройден, значит, пиксел в этой точке ближе к наблюдателю, чем световой шар, т. е. он не может быть освещен. Поэтому в этих точках буфер трафарета уменьшаем на единицу. Пиксельный шейдер по-прежнему остается установленным в NULL.

Далее необходимо еще раз вывести шар (световой шар) с радиусом действия источника света. Второй этап алгоритма иллюстрирует

рис. 4.3. Зеленым цветом показаны пикселы, для которых тест глубины не прошел и значение буфера трафарета было уменьшено на единицу. Оставшиеся красные участки – это пикселы, для которых значение буфера трафарета равно единице. Именно для них на следующем этапе и будет произведен расчет освещения.

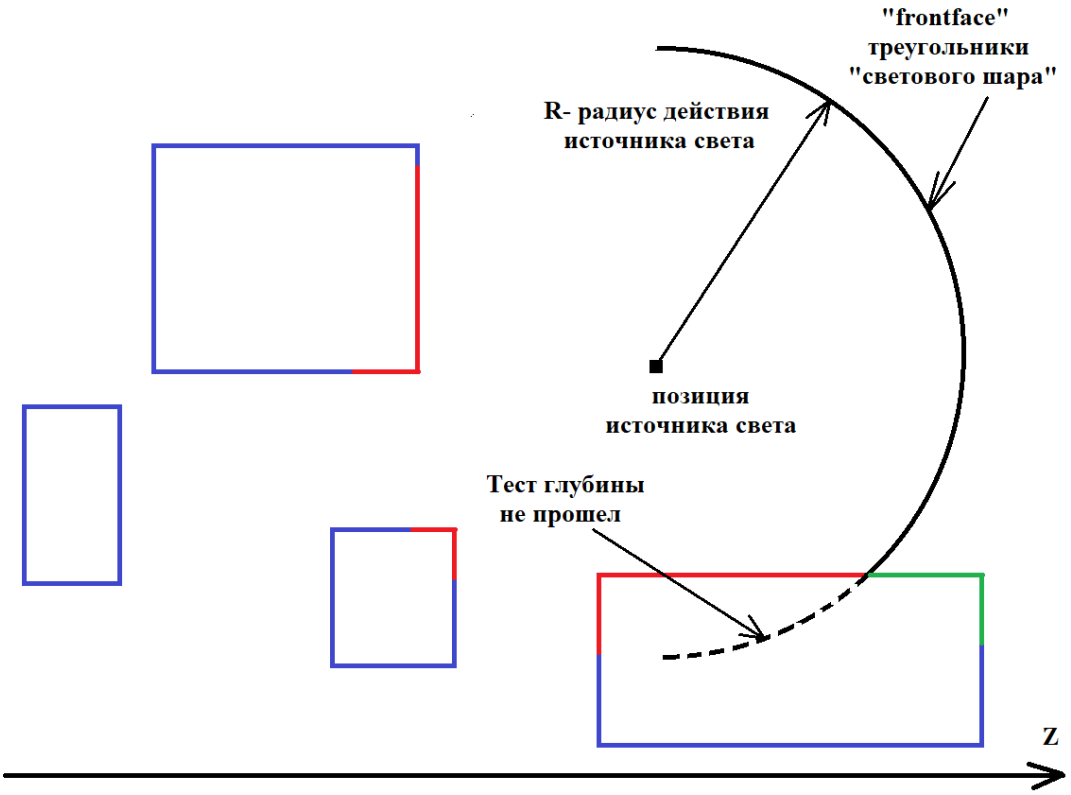


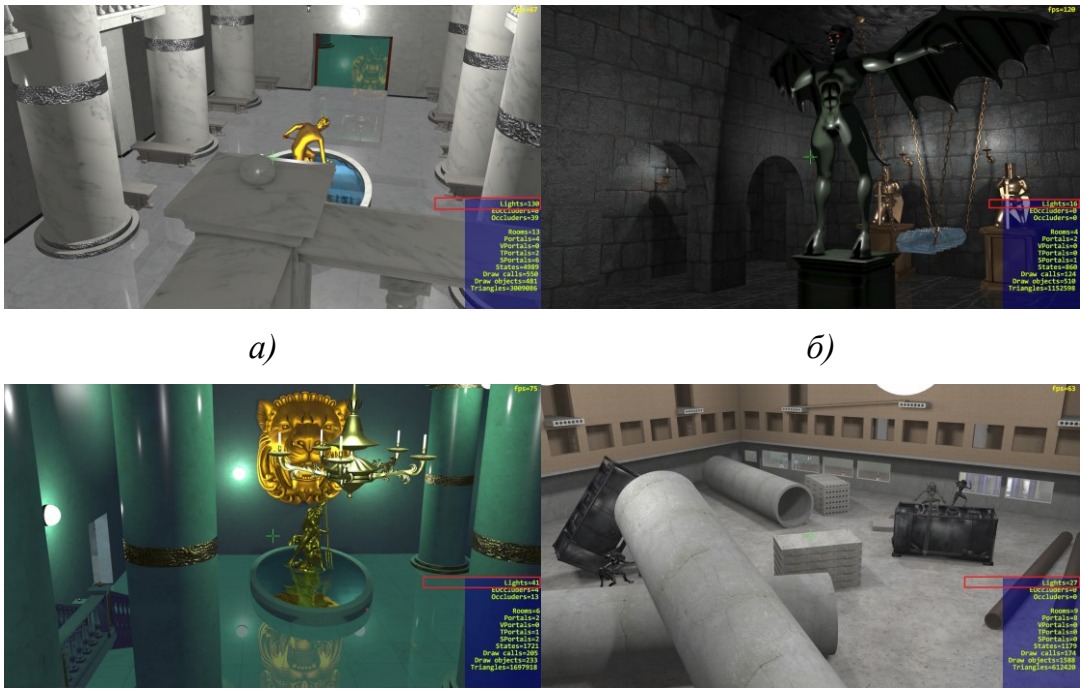
Рис. 4.3. Второй этап алгоритма освещения

3. Передать в пиксельный шейдер характеристики источника света. Включить вывод на поверхность отображения (поле RenderTargetWriteMask структуры состояния смешивания установить в 0x0f). Включить режим смешивания и установить настройки смешивания, как это было показано ранее в этом параграфе. Состояние растеризации оставить прежним (выводим только «лицевые» треугольники). Выключить буфер глубины (значение поля DepthEnable приравнять к нулю). Заполнить поля структуры FrontFace

```
FrontFace.StencilFailOp = D3D11_STENCIL_OP_KEEP;
FrontFace.StencilDepthFailOp = D3D11_STENCIL_OP_KEEP;
FrontFace.StencilPassOp = D3D11_STENCIL_OP_DECR;
FrontFace.StencilFunc = D3D11_COMPARISON_EQUAL;
```

Значение value необходимо оставить равным единице. Установить шейдер расчета освещенности по одной из моделей, описанных в гл. 3. И в третий раз вывести «световой шар». Расчет освещения будет произведен только тогда, когда тест трафарета пройден, следовательно, только для тех пикселов, которые находятся внутри «светового шара», т. е. должны быть освещены. И для этих же пикселов значение буфера трафарета будет уменьшено на единицу, т. е. значение в буфере трафарета вернется к нулю.

После этих шагов можно либо вернуться к п. 1 алгоритма для расчета освещения другим источником, либо восстановить настройки состояний растеризации, смешивания и буфера глубины и трафарета. Таким образом, первые два шага алгоритма выделяют те пикселы, которые будут освещены, а на третьем шаге происходит расчет освещения. Как можно увидеть на рис. 4.4, *а – г*, применение метода отложенного освещения и оптимизации расчетов освещения позволяет отображать трехмерные сцены с количеством источников света вплоть до сотен. На рисунках показаны трехмерные сцены соответственно со 130 (рис. 4.4, *а*), 16 (рис. 4.4, *б*), 41 (рис. 4.4, *в*) и 27 (рис. 4.4, *г*) источниками света.



а) *б)*

в) *г)*

Рис. 4.4. Трехмерные сцены с несколькими источниками света

Аналогичный алгоритм можно применять и для направленных точечных источников. Для не точечных направленных источников необходимо освещать всю сцену, т. е. выводить четырехугольник по размеру рабочей области окна с шейдером расчета освещения.

Для точечных направленных источников света, схематично изображенных на рис. 3.1, необходимо рассчитать коэффициент ослабления, связанный с тем, что мощность светового потока тем меньше, чем больше угол между вектором света и направлением источника. С другой стороны, чем меньше углы φ и θ, тем мощность светового потока выше за счет фокусировки (по крайней мере это справедливо для фонариков). Код в пиксельном шейдере учитывающий все характеристики такого источника, выглядит следующим образом:

```
// Расчет конуса света
// Phi – внешний угол
// Theta – внутренний угол
// Falloff – степень ослабления с увеличением угла
float3 Direct = normalize(direction.xyz); // Вектор направления
float Sp = 0.0;
float d = saturate(dot(L, Direct));
if (d >= Phi) discard;
if (d < Theta) Sp = 1.0;
if ((d >= Theta) && (d < Phi))
{
        Sp = saturate(d - Phi) / (Theta - Phi);
        Sp /= (Falloff - Sp * Falloff + Sp);
}
float acselrator = PI / (acos(Phi) + acos(Theta));
Sp *= acselrator * acselrator;
```

Еще одной оптимизацией, но весьма спорной, касающейся всех источников освещения, может служить проверка совпадения нормали и вектора света на ранней стадии. Аналогично предложенному алгоритму можно в буфере трафарета отмечать те пикселы, которые точно будут освещены (скалярное произведение нормали пиксела и вектора света больше нуля), и далее еще одним проходом уже выполнять освещение пикселов, отмеченных в буфере трафарета. Однако такая

проверка уже не является аппаратной и не всегда корректна при всех моделях освещения.

Еще один момент, который необходимо запомнить, – это правильная установка состояния фильтрации текстур. Дело в том, что в текстурах, формирующих *G*-буфер, находятся данные, которые искажать нельзя, в противном случае можно получить неверные расчеты освещения. Поэтому необходимо создать состояние фильтрации текстур и установить его в определенный слот шейдера. Для отключения фильтрации необходимо в поле Filter структуры D3D11_SAMPLER_DESC установить значение D3D11_FILTER_MIN_MAG_MIP_POINT, что фактически отключит какую-либо фильтрацию.

Описанные в этом параграфе подходы и алгоритмы производят расчет освещения всех пикселов, попавших в область действия источника света, как показано на рис. 4.5. Поверхности *A* и *B* оказываются освещенными, хотя поверхность *A* должна быть в тени поверхности *B*, т. е. не все пикселы должны быть освещены, поскольку часть из них может быть загорожена другими пикселами или объектами, которые могут находиться ближе к источнику света. Таким образом, изложенные методы не позволяют получить тени.

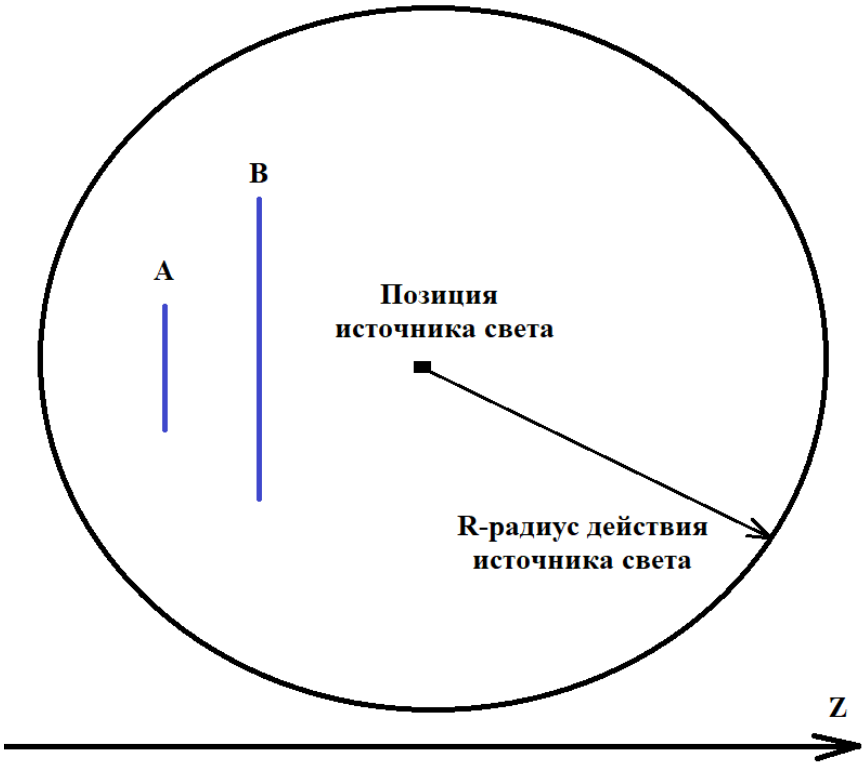


Рис. 4.5. Освещение объектов точечным источником света

Наличие теней в трехмерной сцене значительно улучшает восприятие и, конечно, добавляет реализма. Чем качественнее сделаны тени, тем более реалистично и физически корректно выглядит отображаемая сцена. Поэтому в следующих параграфах будет рассмотрен расчет теней и затенения.

4.3. Затенение окружением

В этом параграфе будет рассмотрен метод затенения окружением SSAO (screen space ambient occlusion). Вообще, этот метод не является физически корректным, он лишь имитирует затенение. Причем эта имитация относится скорее к мелким деталям объектов, для которых расчет теней может оказаться очень ресурсозатратным. Пример затенения окружением можно увидеть на рис. 4.6. На рис. 4.6, *а* затенение отключено, а на рис. 4.6, *б* применен метод SSAO.

а) *б)*

Рис. 4.6. Пример метода SSAO

Метод SSAO основан на последующей обработке уже нарисованной трехмерной сцены и использовании данных из G-буфера. Суть метода, хорошо описанная в статье [9], заключается в том, что после отображения трехмерной сцены вокруг пиксела, для которого рассчитывается затенение (на рис. 4.7 такие пикселы показаны красным цветом), находятся другие пикселы, которые могут его частично закры-

вать от рассеянного освещения. И наоборот, некоторые пикселы не затеняются соседними (на рис. 4.7 показаны зеленым цветом). Причем чем ближе оказываются затеняющие пикселы к затеняемому пикселу, для которого проводится расчет, тем сильнее затенение. Кроме того, интуитивно понятно, что чем «выше» оказываются затеняющие пикселы, тем сильнее должно быть затенение.

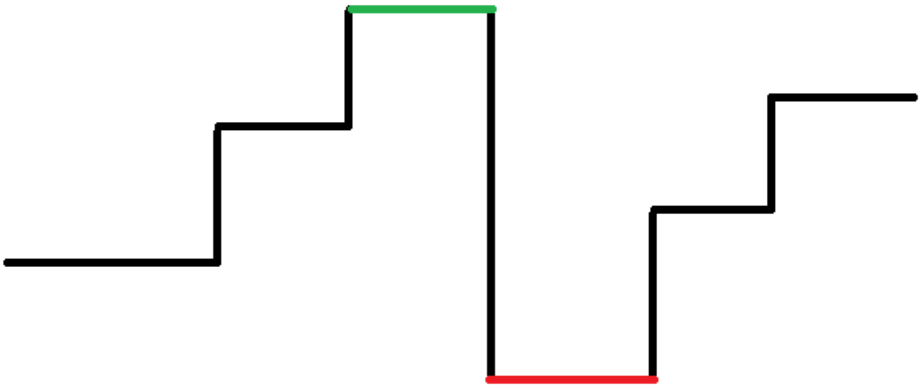


Рис. 4.7. Затенение соседними пикселами

Понятие «выше», конечно, не является корректным, но на рис. 4.8 поясняется, о чем идет речь. Дело в том, что затеняемый пиксел обладает собственной нормалью, и интуитивно понятно, что чем больше скалярное произведение векторов N и V, показанных на рис. 4.7, и чем меньше длина вектора V, тем сильнее затенение.

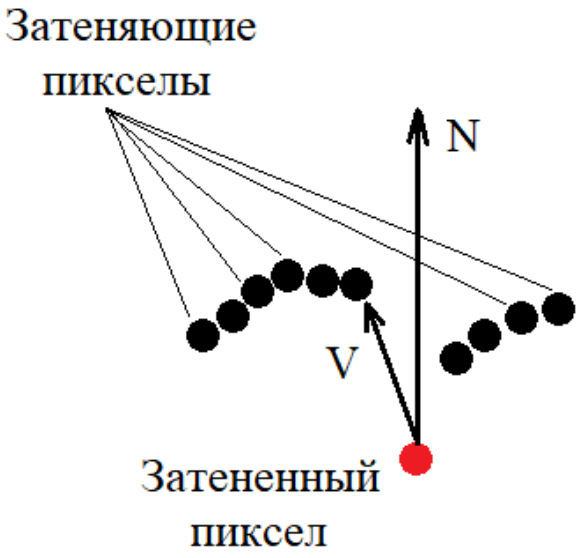


Рис. 4.8. Затенение соседними пикселами

Можно предложить функцию для вычисления затенения, например:

$$Occlusion = \max(0.0, dot(N,V))*(1.0/(1.0+d)), \quad (2)$$

где *Occlusion* – рассчитываемое затенение; *d* – длина вектора *V*; *N*-нормаль из G-буфера. Множитель $(1.0/(1.0+d))$ выбран таким [9], чтобы снизить линейную зависимость от расстояния. В принципе, можно выбирать и квадратичную зависимость от расстояния или какие-либо другие функции.

Таким образом, алгоритм SSAO сводится к выбору соседних пикселов и суммированию уровня затенения от каждого из них с последующим усреднением по количеству выборок. Для реализации алгоритма SSAO необходимо создать еще одну поверхность рендеринга, как это было показано в предыдущих параграфах, отключить буфер глубины, отключить фильтрацию текстур, установить текстуры G-буфера в соответствующие слоты и вывести четырехугольник по размеру экрана с пиксельным шейдером, который и будет осуществлять алгоритм SSAO.

Поскольку в пиксельном шейдере потребуется не только позиция затеняемого пиксела, но и позиции затеняющих пикселов, создадим функцию восстановления позиции (на основе функции WSPosition(pos), приведенной в прил. 5), которая возвращает не только позицию пиксел в мировом пространстве, но и значение из буфера глубины для этого пиксела.

```
float4 GetPosition(in float2 tuv)
{
    float z= Depth.Sample(Sample1, tuv).r;
    return float4(WSPosition(float3(uv,z)), z);
}
```

Функция расчета затенения будет выглядеть следующим образом [9]:

```
float AmbientOcclusion(float2 coord, float3 p, float3 normal,float4 ssao)
{
    float3 diff = GetPosition(coord).xyz - p;
    const float3 v = normalize(diff);

    const float d = length(diff)*ssao.z;
    return max(0.0, dot(cnorm, v) - ssao.y)/(1.0+d*d)*ssao.w;
},
```

где coord – координаты на экране затеняющего пиксела; p – позиция в мировом пространстве затеняемого пиксела; normal – его нормаль; ssao – коэффициенты настроек затенения по методу SSAO, о которых будет рассказано ниже. Функция достаточно простая. Сначала получаем позицию в трехмерном пространстве затеняющего пиксела, далее считаем затенение по формуле (2).

Осталось определить, как выбирать пикселы для расчета затенения. Дело в том, что чем меньше затеняющих пикселов будет выбрано, тем быстрее будет производиться расчет. С другой стороны, недостаточный объем выборок даст неверную картину. Поэтому целесообразно выборки делать случайным образом и равномерно вокруг затеняемого пиксела. Даже если что-то важное будет пропущено, то случайность даст возможность не пропустить это при расчете затенения для соседнего пиксела.

Хороший алгоритм выборок предложен в статье [9]. Для этого алгоритма используется специальная текстура, как на рис. 4.9, в которой содержатся случайно направленные векторы (вообще, они трехмерные и направлены по полусфере, в направлении наблюдателя, но в нашем случае потребуются только компоненты *x* и *y* этих векторов).

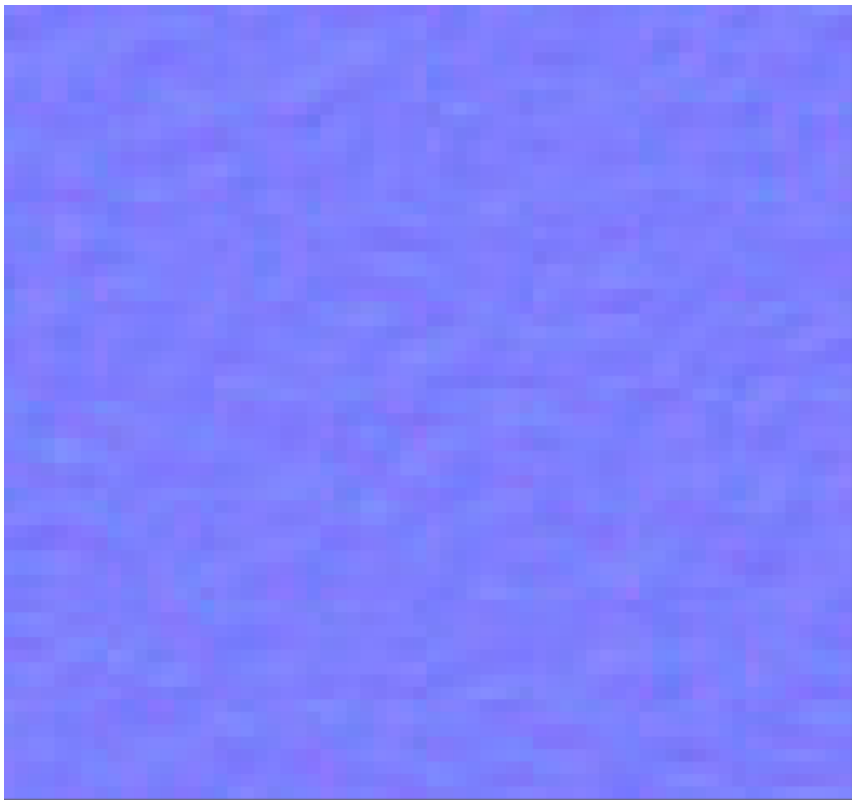


Рис. 4.9. Текстура со случайными векторами

В пиксельном шейдере добавим функцию получения случайного вектора

```
float2 GetRandom(float2 uv)
{
    return normalize(Random.Sample(Sample1, uv * resolution.zw / random_size).xy * 2.0f - 1.0f);
},
```

где resolution.zw – разрешение экрана; random_size – разрешение текстуры со случайными векторами.

Теперь можно создать массив из четырех единичных векторов на плоскости, каждый из которых повернут на 90° относительно предыдущего

```
const float2 vec[4] = { float2(1.0,0.0), float2(0.0,1.0), float2(-1.0,0.0), float2(0.0,-1.0) };
```

Рассчитав вектор отражения следующим образом:

```
float2 rand = GetRandom(pos);
float2 coord1 reflect(vec[j], rand);
```

можно получить четыре вектора, повернутых случайно, как показано на рис. 4.10.

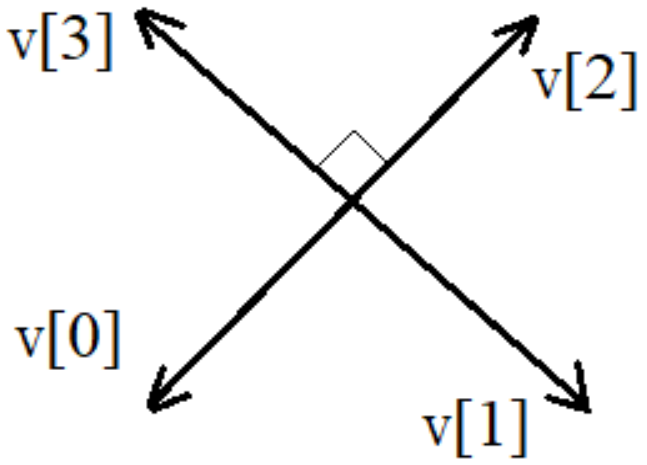


Рис. 4.10. Поворот векторов

Выборки осуществляются в цикле следующим кодом:

```
float2 rand = GetRandom(pos);
int iterations = 4;
for (int j = 0; j < iterations; ++j)
```

```
{
float2 coord1 reflect(vec[j], rand) *rad;
float2 coord2 = float2(coord1.x*0.707 - coord1.y*0.707,
coord1.x*0.707 + coord1.y*0.707);
        ao += AmbientOcclusion(pos + coord1*0.25, p.xyz, n,ssao);
        ao += AmbientOcclusion(pos + coord2*0.5, p.xyz, n,ssao);
        ao += AmbientOcclusion(pos + coord1*0.75, p.xyz, n,ssao);
        ao += AmbientOcclusion(pos + coord2, p.xyz, n,ssao);
}
```

Вектор coord2 – это поворот векторов на рис. 4.10 на 45°. Переменная rad – это радиус, внутри которого происходят выборки. Радиус задается как один из параметров алгоритма SSAO. Кроме того, для учета перспективы радиус делится на расстояние от камеры до пиксела. Это расстояние от камеры до пиксела возвращается функцией GetPosition(), рассмотренной выше. Код расчета радиуса

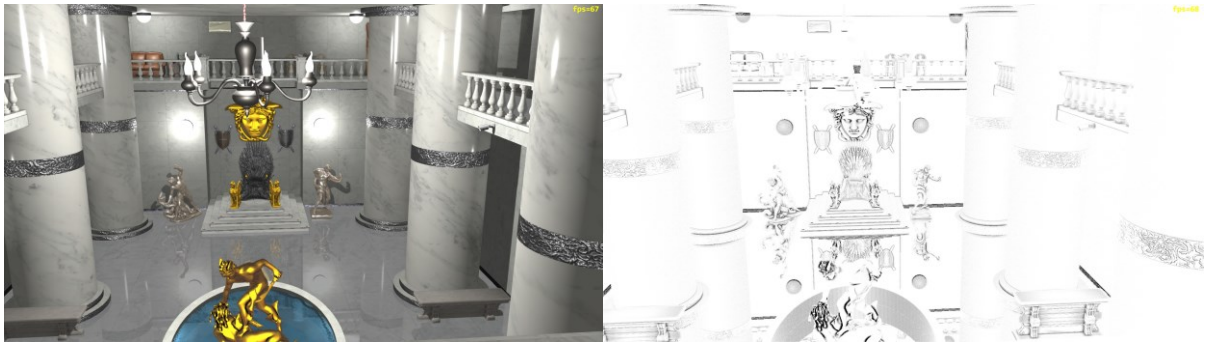
```
float4 p = GetPosition(pos);
// Линеаризовать глубину для диапазона от 0 до 1
float z =p.w * 2.0 - 1.0;
float depth= (2.0 * near * far) / (far + near - z * (far - near));
depth = (depth - near) / (far - near);
float rad = ssao.x / depth;
```

В этом коде происходит линеаризация глубины удаления пиксела от камеры и приведение значений глубины к диапазону от 0 до 1. Линеаризация дает лучшие результаты, чем использование нелинейных значений из буфера глубины. Для линеаризации в шейдер посредством константного буфера необходимо передать минимальное (near) и максимальное (far) расстояния от камеры, которые применялись при создании матрицы перспективной проекции.

Итак, в шейдере осуществляется 16 выборок, и окончательный расчет затенения выглядит так:

```
ao /= (float)iterations*4.0;
ao = max(0,1.0 - ao);
return float4(ao, ao, ao, 1.0);
```

Полный текст пиксельного шейдера, производящего расчет затенения по методу SSAO, приведен в прил. 6. На рис. 4.11 показан результат выполнения шейдера затенения по методу SSAO для трехмерной сцены (рис. 4.11, *а*). Полученную текстуру (рис. 4.11, *б*) теперь можно использовать, перемножая результаты расчета освещения и затенения. Удобнее это делать в каком-нибудь финальном шейдере, который, например, проводит гамма-коррекцию.



а) *б)*

Рис. 4.11. Результат выполнения шейдера затенения по методу SSAO

Необходимо отметить, что метод SSAO не лишен недостатков. Наилучшего результата затенения можно добиться только тщательным подбором параметров константы SSAO в шейдере. Константа имеет четыре компоненты, каждая из которых влияет на работу алгоритма SSAO

ssao.x – радиус, внутри которого происходит выборка;

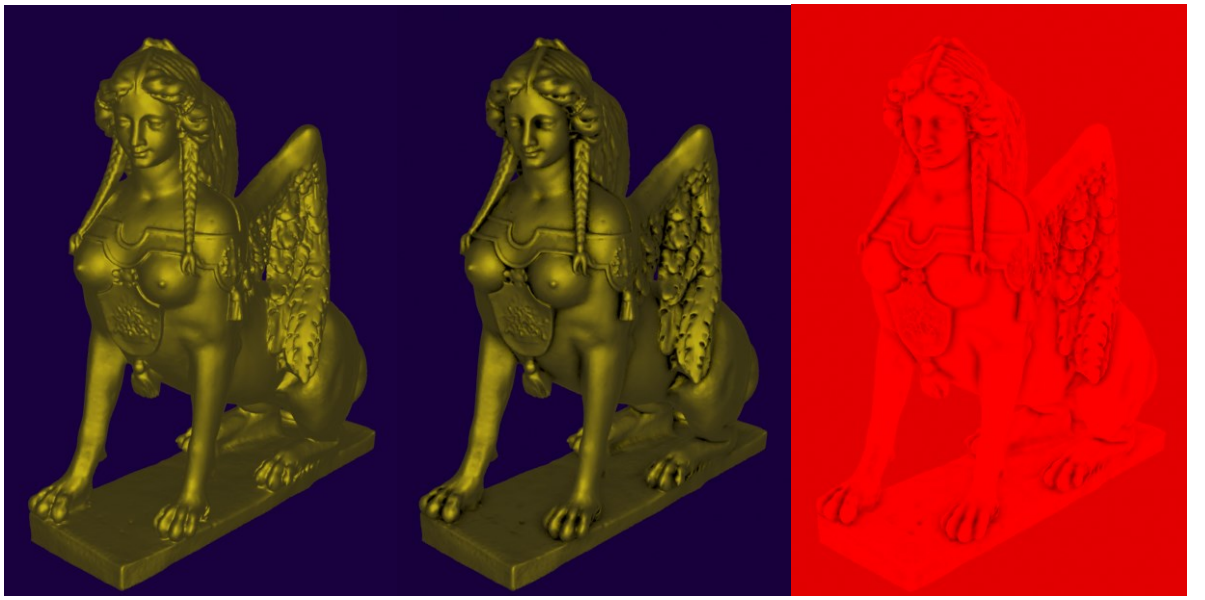
ssao.y – параметр, который управляет конусом выборки, т. е. скалярное произведение векторов N и V, меньшее, чем это значение, не учитывается в затенении;

ssao.z – масштабирует расстояние, т. е. длину вектора V;

ssao.w – коэффициент интенсивности затенения.

Поскольку в разных сценах или для разных объектов значения этих параметров могут различаться, можно предложить считать набор этих параметров еще одной характеристикой материала. В этом случае всегда можно создать для нужного объекта материал с необходимыми коэффициентами затенения.

На рис. 4.12, *а* – *е* показаны примеры затенения объекта с разным выбором параметров затенения.



а) *б)* *в)*

г) *д)* *е)*

Рис. 4.12. Различные параметры затенения методом SSAO: а – без затенения; б – SSAO (0,075; 0; 5; 3); в – SSAO (0,05; 0; 5; 1); г – SSAO (0,075; 0; 5; 3); д – SSAO (0,075; 0,25; 5; 3); е – SSAO (0,075; 0; 50; 3)

Остается добавить, что при достаточно большом увеличении можно наблюдать некоторую «зернистость» теней. Это связано с тем, что выборки осуществляются случайно. Чтобы снизить получившийся негативный эффект, можно после формирования текстуры с затенением посоветовать применение какого-нибудь шейдера со сглаживающим фильтром, чтобы тени выглядели менее зернисто.

4.4. Тени от источников света

В п. 4.2 был рассмотрен вопрос применения отложенного освещения. В качестве одного из недостатков указывалось освещение всех без исключения объектов, попавших в область действия источника света, т. е. отсутствие теней, которые могут отбрасывать объекты. Конечно, наличие теней от объектов значительно улучшит визуальное восприятие трехмерной сцены и приблизит нас к физически корректному освещению. На рис. 4.13, *а – г* показаны трехмерные сцены без теней и с тенями.

а) *б)*

в) *г)*

Рис. 4.13. Тени в трехмерных сценах

В 1978 году Ланс Уильямс предоставил алгоритм создания теней, который назывался projective shadowing (shadow mapping). С тех пор встречается много вариаций этого метода построения теней, но принцип остается неизменным.

Алгоритм заключается в том, что перед выводом трехмерной сцены для каждого источника света с его позиции и/или в его направлении устанавливают камеру и производят рендеринг трехмерной сцены с той лишь разницей, что вывод цветовой составляющей игнорируют, интерес представляетлишь лишь буфер глубины. В буфере глубины будет содержаться, по сути, расстояние до источника света. Полученную текстуру называют *картой теней*. Затем уже при рендеринге сцены, в процессе освещения, пространственные координаты каждого пиксела переносят в пространство источника света, определяют их координаты на карте теней, рассчитывают расстояние до источника и сравнивают со значением из карты теней. Если значение меньше, пиксел освещается, если нет, значит, он в тени и освещение пиксела этим источником отсутствует.

Для разных типов источников света реализация этого алгоритма будет несколько отличаться. В настоящем учебном пособии будет рассмотрена реализация алгоритма построения теней для точечных всенаправленных источников света, которая была хорошо рассмотрена в статье [10].

Поскольку точечный всенаправленный источник светит во все стороны, то потребуется карта теней со всех возможных направлений. Один из вариантов решения – это создание кубической текстуры (подобные текстуры уже использовались в п. 3.7) и рендеринг окружающего мира на каждую из поверхностей этой текстуры. В данном случае потребуется создание матрицы проекции с углами обзора по высоте и ширине в 90°. Матрица проекции создается функцией

XMMatrixPerspectiveFovLH(XM_PIDIV2, 1.0f, nearZ, farZ)),

где nearZ, farZ – ближняя и дальняя плоскости отсечения для построения карты теней. Это важные параметры, и от их значений может зависеть качество полученных теней, потому что значения в карте теней – это значения из буфера глубины, а они не являются линейными и зависят как раз от этих параметров тоже. Целесообразно выбирать значение farZ, равное радиусу действия источника света, а значение nearZ хранить как одну из характеристик источника света или как размер самого источника света.

Кроме матрицы проекции необходимо создать шесть матриц вида для шести камер, направленных в разные стороны из центра куба. Поскольку кубические текстуры имеют строгую очередность порядка сторон, то массивы векторов для каждой камеры следующие:

```
XMFLOAT3 at[6]={
XMFLOAT3(1.0f, 0.0f, 0.0f),XMFLOAT3(-1.0f, 0.0f, 0.0f),
XMFLOAT3(0.0f, 1.0f, 0.0f),XMFLOAT3(0.0f, -1.0f, 0.0f),
XMFLOAT3(0.0f, 0.0f, 1.0f),XMFLOAT3(0.0f, 0.0f, -1.0f),
};
XMFLOAT3 up[6]={
XMFLOAT3(0.0f, 1.0f, 0.0f),XMFLOAT3(0.0f, 1.0f, 0.0f),
XMFLOAT3(0.0f, 0.0f, -1.0f),XMFLOAT3(0.0f, 0.0f, 1.0f),
XMFLOAT3(0.0f, 1.0f, 0.0f),XMFLOAT3(0.0f, 1.0f, 0.0f),
};
```

Позиция камеры совпадает с позицией источника света. Необходимо создать шесть матриц вида (как было показано в п. 2.8) и перемножить их с матрицей проекции. Таким образом, получим массив из шести матриц этих произведений.

Далее необходимо создать кубическую текстуру, которая является буфером глубины

```
ID3D11ShaderResourceView* ShadowDepthTexture=NULL;
ID3D11DepthStencilView* ShadowDepthView=NULL;
D3D11_TEXTURE2D_DESC depthBufferDesc;
D3D11_DEPTH_STENCIL_VIEW_DESC depthStencilViewDesc;
ID3D11Texture2D* pointer = NULL;
ZeroMemory(&depthBufferDesc, sizeof(depthBufferDesc));
depthBufferDesc.Width = resolution;
depthBufferDesc.Height = resolution;
depthBufferDesc.ArraySize = 6;
depthBufferDesc.MipLevels = 1;
depthBufferDesc.Format = DXGI_FORMAT_R16_TYPELESS;
depthBufferDesc.SampleDesc.Count = 1;
depthBufferDesc.SampleDesc.Quality = 0;
depthBufferDesc.Usage = D3D11_USAGE_DEFAULT;
depthBufferDesc.BindFlags =
D3D11_BIND_SHADER_RESOURCE |
D3D11_BIND_DEPTH_STENCIL;
```

```
        depthBufferDesc.CPUAccessFlags = 0;
        depthBufferDesc.MiscFlags =
D3D11_RESOURCE_MISC_TEXTURECUBE;
        if (FAILED(device->CreateTexture2D(&depthBufferDesc, NULL,
&pointer))) return E_FAIL;
        // Ресурс шейдера для буфера глубины
        D3D11_SHADER_RESOURCE_VIEW_DESC descSRV;
        ZeroMemory(&descSRV, sizeof(descSRV));
        descSRV.Format = DXGI_FORMAT_R16_UNORM;
        descSRV.ViewDimension =
D3D11_SRV_DIMENSION_TEXTURECUBE;
        descSRV.TextureCube.MipLevels = 1;
        descSRV.TextureCube.MostDetailedMip = 0;
        if (FAILED(device->CreateShaderResourceView(pointer,
&descSRV, &ShadowDepthTexture))) return E_FAIL;
        ZeroMemory(&depthStencilViewDesc,
sizeof(depthStencilViewDesc));
        depthStencilViewDesc.Format = DXGI_FORMAT_D16_UNORM;
        depthStencilViewDesc.ViewDimension =
D3D11_DSV_DIMENSION_TEXTURE2DARRAY;
        depthStencilViewDesc.Texture2DArray.ArraySize = 6;
        depthStencilViewDesc.Texture2DArray.FirstArraySlice = 0;
        depthStencilViewDesc.Texture2DArray.MipSlice = 0;
        depthStencilViewDesc.Flags = 0;
        // Буфер глубины
        if (FAILED(device->CreateDepthStencilView(pointer, depthSten-
cilViewDesc, &ShadowDepthView))) return E_FAIL;
        ReleaseCOM(pointer);
```

В принципе, такой текстуры достаточно. Вся необходимая информация будет находиться в ней. Но в этой текстуре дальность записана не в линеаризированном виде, и в дальнейшем, определяя, находится ли пиксел в тени, придется сравнивать расстояние от пиксела до источника света и переводить значение глубины из карты теней в линейное пространство. Чтобы избежать лишних операций, можно создать еще одну кубическую тектстуру, куда будет записано уже линейное значение расстояния от источника света до объекта. Код создания такой текстуры выглядит следующим образом:

```
ID3D11ShaderResourceView* ShadowCubeTexture=NULL;
ID3D11RenderTargetView* ShadowTargetView=NULL;
D3D11_TEXTURE2D_DESC desc;
desc.ArraySize = 6;
desc.BindFlags = D3D11_BIND_SHADER_RESOURCE |
D3D11_BIND_RENDER_TARGET;
desc.CPUAccessFlags = 0;
desc.Format = DXGI_FORMAT_R16_UNORM;
desc.Height = resolution;
desc.MipLevels = 1;
desc.MiscFlags = D3D11_RESOURCE_MISC_TEXTURECUBE;
desc.SampleDesc.Count = 1;
desc.SampleDesc.Quality = 0;
desc.Width = resolution;
desc.Usage = D3D11_USAGE_DEFAULT;
if (FAILED(device->CreateTexture2D(&desc, NULL, &pointer)))
return E_FAIL;
if (FAILED(device->CreateShaderResourceView(pointer, NULL, &
ShadowCubeTexture))) return E_FAIL;
if (FAILED(device->CreateRenderTargetView(pointer, NULL,
&ShadowTargetView))) return E_FAIL;
ReleaseCOM(pointer);
```

Еще один важный параметр при создании карты теней – значение переменной resolution, которая задает размер каждой грани текстуры в пикселах. Чем больше это значение, тем точнее окажется карта теней, но тем больше места она будет занимать и, кроме этого, тем дольше карта теней будет строиться. На размер текстуры влияет еще параметр desc.Format, который в данном примере выбран 16-битным.

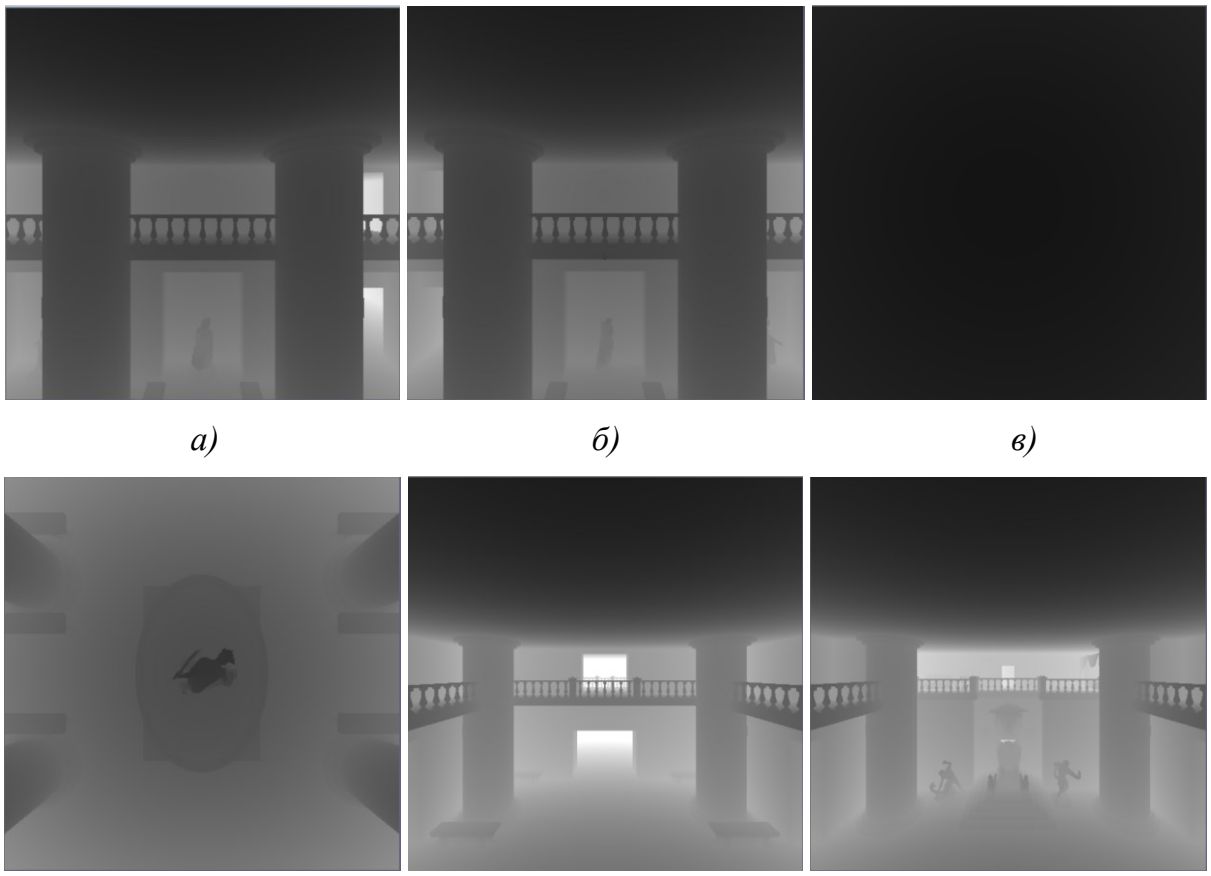
Ранее в подобных алгоритмах использовалось шесть проходов рендеринга для каждой грани куба и с соответствующей матрицей вида. Но возможности DirectX 11 таковы, что можно ограничиться одним проходом, просто используя геометрический шейдер, который вместо одного треугольника будет генерировать шесть треугольников для каждой поверхности гипотетического куба.

Особенность вершинного шейдера заключается в том, что он использует только преобразования вершин, связанные с мировой мат-

рицей. Другими словами, он не использует перевод объекта из мирового пространства в экранное. Это будет делать геометрический шейдер. В прил. 7 приведен код вершинного, геометрического и пиксельного шейдеров для генерации карты теней.

Перед началом создания карты теней необходимо очистить буфер глубины, передать матрицу произведения матрицы вида на матрицу проекции в константный буфер и установить последний для геометрического шейдера. Также установить необходимые шейдеры и вывести все объекты трехмерной сцены.

В результате получится кубическая карта, представленная шестью своими поверхностями на рис. 4.14, *а* – *е*, где поверхности последовательно соответствуют направлениям осей X, $-X$, Y, $-Y$, Z, $-Z$. Поверхность рис. 4.14, *в* абсолютно темная, поскольку источник света находится под потолком и в этом направлении кубической карты расстояния от источника света до потолка очень маленькие.



а) *б)* *в)*

г) *д)* *е)*

Рис. 4.14. Карта теней

После того как для источника света создана карта теней, ее можно использовать, установив как ресурс пиксельного шейдера и в пиксельном шейдере освещения объявив, например, так:

TextureCube<float> Shadow: register(t6);

При расчете освещения с затуханием рассчитывался вектор между пикселом и источником света и расстояние между ними. Теперь это расстояние необходимо перевести в пространство кубической карты теней источника света, но поскольку неизвестно, какая из шести матриц вида использовалась, необходимо преобразовать это расстояние следующим образом:

```
float3 AbsVec = abs(vec);
float LocalZ = max(AbsVec.x, max(AbsVec.y, AbsVec.z));
float Sd=LocalZ/ attenuation.z; // делить на радиус действия источника света
```

В переменной *Sd* находится расстояние от пиксела до источника света в пространстве источника света, вернее, в пространстве его кубической карты теней. Остается получить выборку из кубической текстуры и сравнить со значением *Sd*. Если *Sd* больше, чем значение из текстуры, значит, пиксел не освещается источником света.

Теперь необходимо рассмотреть ряд проблем, возникающих при построении теней. Первая проблема связана с появление муара на всех объектах, как показано на рис. 4.15.

а) *б)*

Рис. 4.15. Самозатенение поверхности

Это вызвано тем, что карта теней дискретна и все объекты и поверхности на этой карте выглядят в виде ступенек, как показано на рис. 4.16. Дискрет карты теней *d* зависит от ее размера, но тем не менее он не бесконечно мал. А вот часть пикселов на поверхности, как видно из рисунка, в реальности оказывается или ближе, чем расстояние, записанное в карту теней (*r*1…*r*8), или дальше, что приводит к явлению, которое называют *самозатенением*.

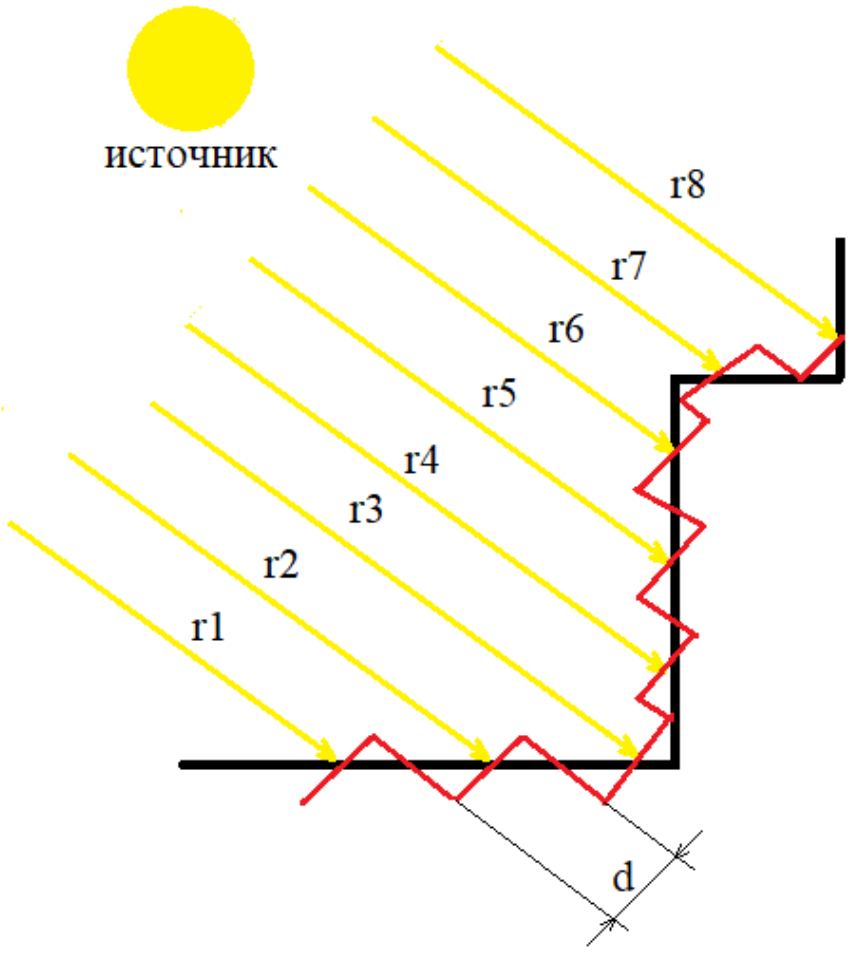


Рис. 4.16. Дискретность карты теней

Избежать подобного можно, если при сравнении глубин увеличивать глубину в карте теней на небольшое расстояние. Правда, это приращение зависит от угла между поверхностью и вектором направления на источник света.

Наилучший вариант – перед рендерингом карты теней создать и установить состояние растеризации, в котором подобрать значения в полях DepthBias, DepthBiasClamp, SlopeScaledDepthBias структуры D3D11_RASTERIZER_DESC. Излишне увеличивать значение приращения глубины тоже не следует, потому что можно получить эффект «Питера Пена», когда тень «оторвана» на некоторое расстояние от своего объекта, как это показано на рис. 4.17.

Рис. 4.17. Эффект «Питера Пена»

Дискретность карты теней создает еще одну проблему, связанную со ступенчатостью границы теней, как это показано на рис. 4.18. Однако DirectX 11 поддерживает аппаратный PCF-фильтр (percentage-closer filtering). Фактически это напоминает билинейную фильтрацию текстур (где вместо конкретного пиксела текстуры берется некоторый усредненный цвет соседних пикселов). Разница в том, что в данном случае происходит сравнение глубин.

а)

б)

Рис. 4.18. Ступенчатая граница тени

Для того чтобы применить PCF-фильтрацию, необходимо создать и установить состояние фильтрации тестур в один из регистров *s* пиксельного шейдера. Следующие поля структуры D3D11_SAMPLER_DESC необходимо заполнить так:

```
desc.Filter=
D3D11_FILTER_COMPARISON_MIN_MAG_MIP_LINEAR;
desc.AddressU= D3D11_TEXTURE_ADDRESS_CLAMP;
desc.AddressV= D3D11_TEXTURE_ADDRESS_CLAMP;
desc.ComparisonFunc=D3D11_COMPARISON_LESS;
```

А в пиксельном шейдере использовать функцию:

```
TextureCube<float> Shadow: register(t6);
SamplerComparisonState Sample2: register(s2);
float k=Shadow.SampleCmpLevelZero(Sample2, L, Sd).r;
```

В переменной *k* находится коэффициент затенения; вектор *L* – вектор, направленный из пиксела на источник света; переменная *Sd* содержит значение глубины пиксела в пространстве источника света.

Результат применения PCF-фильтра приведен на рис. 4.19 для укрупненного фрагмента тени. На рис. 4.19, *а* тени без фильтрации, на рис. 4.19, *б* тени с применением фильтра PCF.

а)

б)

Рис. 4.19. Применение аппаратного фильтра PCF

Применение фильтра PCF делает тени мягче и уже не с такой ярко выраженной ступенчатостью. Для улучшения фильтрации можно предложить не одну выборку из карты теней, а несколько, смещенных по разным направлениям на некоторый радиус, с последующим усреднением. На рис. 4.20, *а* приведены результаты применения фильтра PCF с пятью выборками PCF-5, который использовался в движке Unreal Engine UE4, а на рис. 4.20, *б* – результат применения фильтра PCF 3 × 3 с девятью выборками.

а)

б)

Рис. 4.20. Применение фильтра PCF для пяти и девяти выборок

На рис. 4.21, *а, б* показаны эти же фильтры по сравнению с отсутствием фильтрации на рис. 4.18, *б*. Конечно, применение фильтра PCF с несколькими выборками значительно улучшает качество тени, однако снижает быстродействие, поэтому при разработке приложений необходимо экспериментировать как с числом выборок, так и с их радиусом. Код фильтров PCF-5 и PCF 3 × 3 приведен в статье [6] и прил. 8.

а)

б)

Рис. 4.21. Фильтры PCF-5 и PCF 3 × 3

При разработке приложений можно рекомендовать настройки тени (такие как DepthBias, DepthBiasClamp, SlopeScaledDepthBias, радиус выборок, nearZ, farZ, размер карты теней, тип фильтра), задавать как характеристики источника света и передавать в виде констант в шейдеры, если это необходимо. Такой подход позволит настраивать тени индивидуально для каждого источника освещения.

Кроме того, следует отметить, что процесс построения карты теней в любом случае занимает время. В сцене с большим количеством источников света в реальном времени реализовать это затруднительно. Можно рекомендовать строить карты теней для всей статической геометрии заранее, а в реальном времени – только для динамических объектов с последующим объединением статических и динамических карт.

Алгоритм построения карт теней для направленных не точечных источников принципиально не отличается от изложенного выше алгоритма. Следует отметить только, что потребуется не перспективная, а ортогональная проекция, и не на кубическую текстуру, а на двумерную. При этом необходимо выбирать позицию камеры так, чтобы пространство построения тени захватывало все видимое пользователем пространство.

В этом параграфе тема визуализации теней и теневых карт рассмотрена очень упрощенно. Существует множество разных подходов, например, построение каскадных теней для направленных источников, теневые объемы, стенсильные тени и многие другие, но они выходят за рамки настоящего учебного пособия.

4.5. Отображение полупрозрачных объектов

Как уже было отмечено выше, один из недостатков алгоритма отложенного освещения – сложность отображения полупрозрачных объектов. Основной подход в отображении полупрозрачных объектов заключается в выводе всей непрозрачной геометрии, освещении и последующем выводе уже полупрозрачных объектов, которые тоже должны быть освещены.

Необходимо учитывать, что полупрозрачные объекты для правильного смешения цветов должны выводиться от дальнего к ближнему. Задача сортировки полупрозрачных объектов в общем случае

простыми методами не решается. Всегда можно представить три треугольника, каждый из которых закрывает часть соседнего. Учитывая, что, возможно, придется выводить полупрозрачные объекты сложной формы, следует обратить внимание на алгоритмы, которые решают вопрос сортировки. Один из таких алгоритмов – это построение двоичного дерева с возможным сечением полигонов в процессе построения. Еще одна группа алгоритмов – это алгоритмы порядко-независимой прозрачности (order-independent transparency, OIT).

Алгоритм порядко-независимой прозрачности в общем виде выглядит следующим образом.

1. Отображение непрозрачной части сцены.

2. Для каждого пиксела полупрозрачного объекта определить необходимость его вывода, основываясь на буфере глубины из п. 1.

3. Для каждого пиксела полупрозрачного объекта, который будет отображен, сохранить его позицию в экранном пространстве, в том числе глубину, цвет, нормаль, спекулярную составляющую, индекс материала и т. п.

4. Провести сортировку для каждой позиции пиксела на экране от дальнего к ближнему.

5. Отобразить (с учетом освещения) пикселы, смешивая их с пикселами ранее отображенной непрозрачной геометрии.

Наибольшую трудность в этом алгоритме вызывают вопросы хранения и сортировки полупрозрачных пикселов. Существует несколько подходов:

– получение сцены по слоям (Depth Piling, Reverse Depth Piling, Dual Depth Piling) [8 – 10] и смешивание полупрозрачных объектов от дальнего к ближнему. Недостаток таких алгоритмов – многократное рисование сцены и хранение промежуточных результатов;

– хранение слоев в текстуре MSAA (Stencil Routed A-Buffer). Недостаток – небольшое количество уровней прозрачности (8 или 16) и отказ от мультисэмплирования, поскольку текстура MSAA уже занята;

– алгоритм OIT с применением связанных списков, который лишен недостатков предыдущих алгоритмов.

Рассмотрим реализацию алгоритма OIT с применением связанных списков, хорошо изложенную в статье [11]. Суть алгоритма заключается в хранении и последующей сортировке связанного списка полупрозрачных пикселов для каждой позиции пиксела на экране.

После сортировки производится смешивание цветов из связанного списка и смешивание результата с пикселом непрозрачной геометрии.

Для хранения информации в связанном списке необходима текстура, совпадающая по размеру с экраном и являющаяся заголовком связанного списка, и буфер для хранения информации (как минимум цвета, глубины и ссылки на предыдущий элемент списка). Кроме того, необходимо иметь счетчик для определения свободной ячейки в буфере.

Принцип алгоритма показан на рис. 4.22. В текстуре, которая является заголовком связанного списка, хранятся адреса в буфере, где начинается (на самом деле заканчивается) связанный список. Числа «–1» показывают, что в эту конкретную позицию пиксела полупрозрачные фрагменты не выводились. Цветные квадраты и числа в них – это порядок вывода полупрозрачных пикселов. После вывода фиолетового пиксела с номером 3 в буфер был записан голубой пиксел с номером 4, а потом – желтый пиксел с номером 5. Но в заголовке списка в этом месте уже ячейка занята, и там находится ссылка на 3-й элемент в буфере. Поэтому ссылка на 3-й элемент записывается в поле ссылок на предыдущий элемент 5-го номера, а в заголовок связанного списка записывается ссылка на 5-й элемент в буфере.

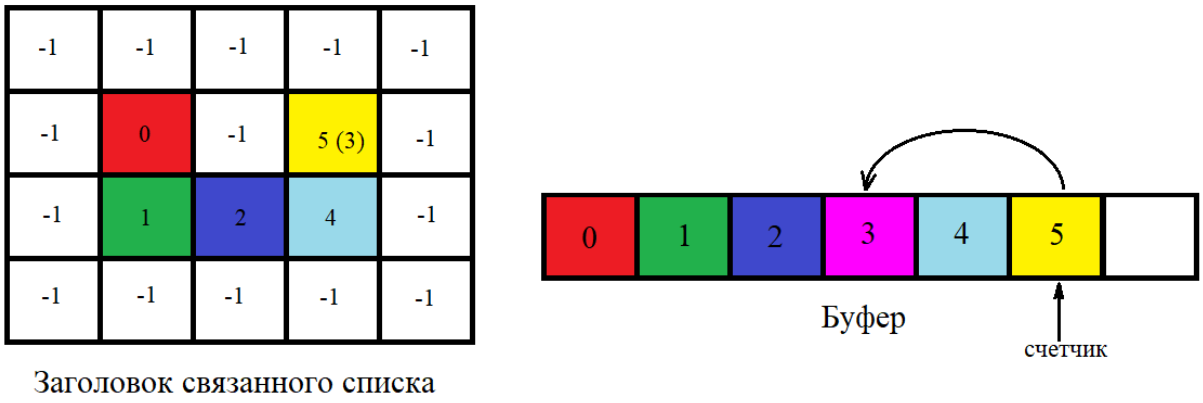


Рис. 4.22. Связанный список

Если теперь будет выведен пиксел по тем же координатам, что и номера 3 и 5, то счетчик увеличится на единицу и в ячейку буфера (на которую теперь указывает счетчик) запишется новый элемент; в его поле ссылок запишется значение 5 (из заголовка списка), а в заголовок списка запишется значение счетчика, т. е. номер 6. Таким обра-

зом, заголовок списка указывает на последний элемент, последний – на предыдущий, и так до тех пор, пока указатель не станет равным –1, т. е. пока список не закончится.

Рассмотрим теперь функции создания и применения заголовка связанного списка и буфера. И заголовок связанного списка, и буфер – ресурсы, в которые можно и писать, и читать из шейдеров (Unordered access). Для таких ресурсов существуют специальные функции, поскольку первые занимают слоты, куда помещаются поверхности рендеринга. Кроме того, в силу высокопараллельной архитектуры GPU работа с такими ресурсами осуществляется с помощью атомарных операций.

Программный код создания заголовка связанного списка выглядит следующим образом:

```
ID3D11UnorderedAccessView* OIT_target=NULL;
ID3D11UnorderedAccessView* OIT_buffer_target=NULL;
ID3D11Texture2D* pointer=NULL;
D3D11_TEXTURE2D_DESC desc;
desc.ArraySize = 1;
desc.BindFlags = D3D11_BIND_UNORDERED_ACCESS |
D3D11_BIND_SHADER_RESOURCE;
desc.CPUAccessFlags = 0;
desc.Format = DXGI_FORMAT_R32_UINT;
desc.Height = windowHeight;
desc.MipLevels = 1;
desc.MiscFlags = 0;
desc.SampleDesc.Count = 1;
desc.SampleDesc.Quality = 0;
desc.Width = windowWidth;
desc.Usage = D3D11_USAGE_DEFAULT;
if (FAILED(device->CreateTexture2D(&desc, NULL, &pointer)))
return
                                                    E_FAIL;
if (FAILED(device->CreateUnorderedAccessView(pointer, NULL,
                                        &OIT_target))) return E_FAIL;
ReleaseCOM(pointer);
```

Поскольку заголовок связанного списка задается по размеру рабочей области окна, то windowWidth и windowHeight – это ширина и высота окна соответственно. Указатели на UNORDERED_ACCESS ресурсы – на заголовок связанного списка и на буфер самого списка – целесообразно хранить в памяти друг за другом, поскольку это облегчит их код подключения к пиксельному шейдеру.

Программный код создания буфера для связанного списка выглядит так:

```
ID3D11Buffer* OIT_buffer=NULL;
UINT size=12;
D3D11_BUFFER_DESC bdesc;
bdesc.ByteWidth = windowWidth * windowHeight *
MAX_WINDOWS_COUNT * size;
bdesc.BindFlags = D3D11_BIND_SHADER_RESOURCE |
D3D11_BIND_UNORDERED_ACCESS;
bdesc.MiscFlags =
D3D11_RESOURCE_MISC_BUFFER_STRUCTURED;
bdesc.StructureByteStride = size;
bdesc.Usage = D3D11_USAGE_DEFAULT;
bdesc.CPUAccessFlags = 0;
if (FAILED(device->CreateBuffer(&bdesc, 0, &OIT_buffer))) return
E_FAIL;
D3D11_UNORDERED_ACCESS_VIEW_DESC sudesc;
sudesc.Format = DXGI_FORMAT_UNKNOWN;
sudesc.ViewDimension = D3D11_UAV_DIMENSION_BUFFER;
sudesc.Buffer.FirstElement = 0;
sudesc.Buffer.Flags = D3D11_BUFFER_UAV_FLAG_COUNTER;
sudesc.Buffer.NumElements = windowWidth * windowHeight *
MAX_WINDOWS_COUNT;
if (FAILED(device->CreateUnorderedAccessView(OIT_buffer,
&sudesc,
&OIT_buffer_target))) return E_FAIL;
ReleaseCOM(OIT_buffer);
```

Переменная size – это размер одного элемента буфра в байтах. Цвет, глубина и ссылка на предыдущий элемент имеют целый 32-разрядный тип, следовательно, переменная size равна 12. Константа MAX_WINDOWS_COUNT определяет, сколько полупрозрачных

объектов размером со всю рабочую область экрана может храниться в буфере. Не стоит выбирать значение NumElements слишком большим, поскольку маловероятно, что в какой-либо трехмерной сцене окажется огромное количество полупрозрачных объектов. Размер буфера при MAX_WINDOWS_COUNT=8 и разрешении экрана 1920 × 1080 составит примерно 190 Мб, а с учетом того что, возможно, потребуется хранить нормаль, индекс материала, спекулярную составляющую, размер одного элемента в буфере и сам буфер могут возрасти вдвое.

Перед тем как заполнять полупрозрачными объектами связанный список, необходимо запретить запись в буфер глубины (но не отключать его), установив нужное состояние буфера глубины и трафарета, запретить вывод на поверхность отображения, установив нужное состояние смешивания (возможно параллельно что-то все же будет выводиться, тогда отключать вывод не следует), очистить заголовок связанного списка и установить для него ресурсы. Следующий код очищает заголовок и устанавливает ресурсы связанного списка:

```
unsigned int clearValue[4] = { 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff };
deviceContext->ClearUnorderedAccessViewUint(OIT_target, clearValue);
unsigned int initvalue[2] = { 0xffffffff,0};
deviceContext->OMSetRenderTargetsAndUnorderedAccessViews
(1, &target, depth_stencil, number, count, UA, initvalue);,
```

где target – поверхность рендеринга; depth_stencil – буфер глубины; number – номер слота рендеринга для заголовка связанного списка (номер слота для буфера будет равен number+1); count – количество unordered access ресурсов; UA – адрес массива из unordered access ресурсов (вот почему указатели на ресурсы желательно располагать рядом в памяти, в нашем случае UA будет равно &OIT_target); initvalue – имя массива с начальными значениями атомарных счетчиков.

После всех приготовлений необходимо вывести полупрозрачные объекты в любом порядке. При этом в пиксельном шейдере необходимо описать unordered access ресурсы

```
struct ListNode
{
	uint packedColor;
	uint depth;
	uint next;
};
globallycoherent RWTexture2D<uint> headBuffer;
globallycoherent RWStructuredBuffer<ListNode> fragmentsList;
```

Поскольку компоненты цвета будут храниться в буфере в виде целого 32-разрядного значения, необходимо будет их упаковать функцией

```
uint packColor(float4 color)
{
	return (uint(color.r * 255) << 24) | (uint(color.g * 255) << 16) |
(uint(color.b * 255) << 8) | uint(color.a * 255);
}
```

Также потребуется модификатор [earlydepthstencil] перед функцией main(…), чтобы сначала прошел тест глубины, а только потом выполнялся шейдер, если тест пройден. В противном случае в буфер могут быть занесены пикселы, которые не будут выведены на экран.

Код добавления нового элемента в буфер глубины и коррекцию указателя в заголовке связанного списка использует атомарные функции

```
uint newHeadBufferValue = fragmentsList.IncrementCounter();
if (newHeadBufferValue == 0xffffffff) return output;
uint2 upos = uint2(input.position.xy);
uint previosHeadBufferValue;
InterlockedExchange(headBuffer[upos], newHeadBufferValue,
previosHeadBufferValue);
```

Функция InterlockedExchange значение headBuffer[upos] записывает в переменную previosHeadBufferValue, а значение переменной newHeadBufferValue, в свою очередь, – в headBuffer[upos].

И напоследок заполняем буфер данными

```
uint currentDepth = f32tof16(input.position.z);
ListNode node;
```

```
node.packedColor = packColor(color_tex);
node.depth = currentDepth;
node.next = previosHeadBufferValue;
fragmentsList[newHeadBufferValue] = node;
```

Следует отметить преобразование значения глубины к 16-разрядному значению. Это связано с экономией места в элементе буфера связанного списка, поскольку, возможно, там необходимо будет хранить и другие данные.

После того как связанный список заполнен, можно приступать к сортировке в связанном списке и смешиванию полупрозрачных фрагментов с непрозрачной геометрией. Для этого необходимо отключить буфер глубины, установив нужное состояние буфера глубины и трафарета, а в настройках состояния смешивания необходимо выбрать следующие параметры:

```
desc.RenderTarget[j].BlendEnable=TRUE;
desc.RenderTarget[j].SrcBlend= D3D11_BLEND_ONE;
desc.RenderTarget[j].DestBlend= D3D11_BLEND_SRC_ALPHA;
desc.RenderTarget[j].BlendOp= D3D11_BLEND_OP_ADD;
desc.RenderTarget[j].SrcBlendAlpha= D3D11_BLEND_ONE ;
desc.RenderTarget[j].DestBlendAlpha= D3D11_BLEND_ZERO;
desc.RenderTarget[j].BlendOpAlpha= D3D11_BLEND_OP_ADD;
desc.RenderTarget[j].RenderTargetWriteMask=15;
```

После этого можно снова воспользоваться функцией OMSetRenderTargetsAndUnorderedAccessViews, установив и ресурсы связанного списка с его заголовком, и поверхность, на которой уже отображена непрозрачная геометрия. Остается установить пиксельный шейдер для сортировки и смешивания и вывести прямоугольник размером с рабочую область окна отображения. Следует отметить, что в шейдере при сортировке полупрозрачных фрагментов по дальности необходимо ограничиться каким-либо значением количества сортируемых фрагментов. От этого будет зависеть, сколько полупрозрачных объектов в сцене может находиться друг за другом.

Полный код шейдеров, реализующих алгоритм OIT на связанных списках, приведен в статье [11] и прил. 9. На рис. 4.23 изображен пример реализации алгоритма OIT на связанных списках. В сцене выведены полупрозрачные фрагменты в виде синего, зеленого и красного квадратов.

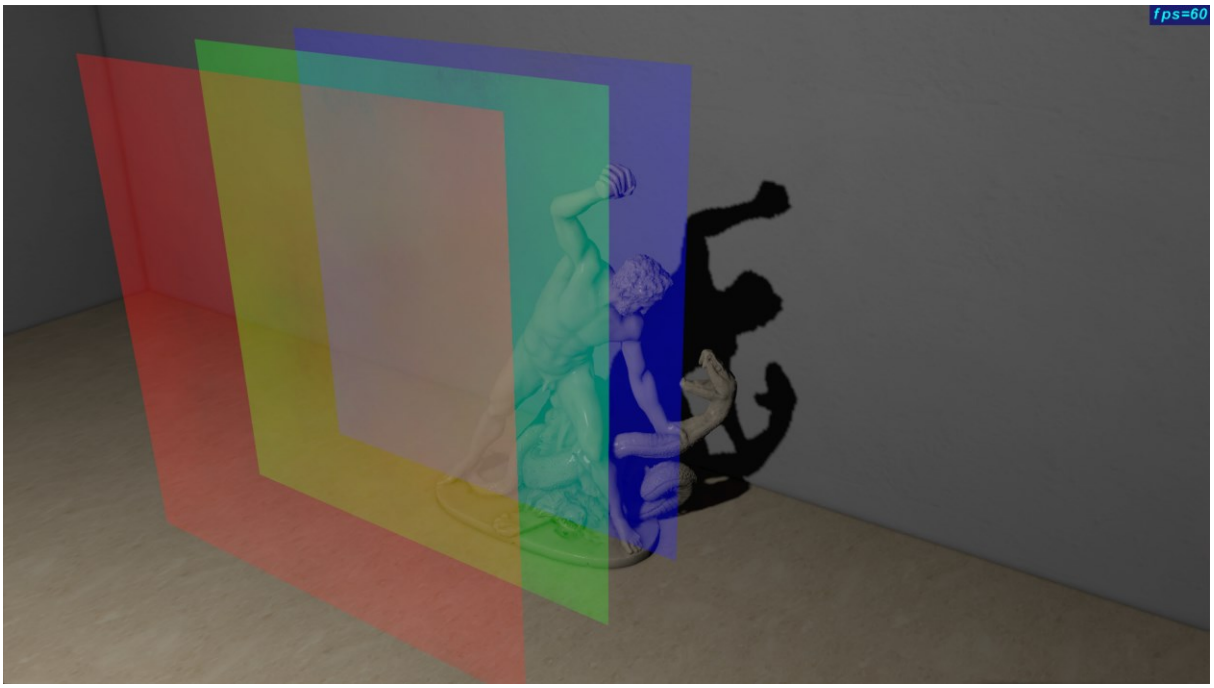


Рис. 4.23. Пример реализации алгоритма OIT на связанных списках

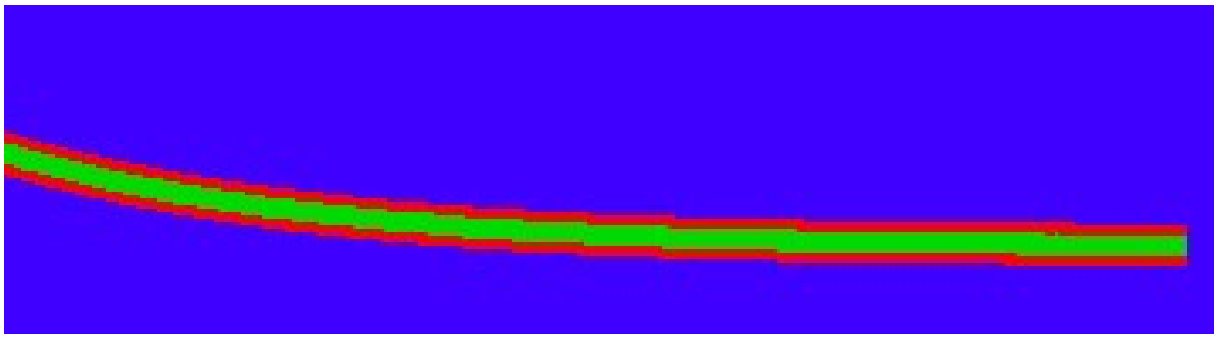
Остается добавить несколько слов об освещении полупрозрачных объектов. Конечно, визуально полупрозрачные объекты с освещением и бликами будут смотреться гораздо эффектнее и правдоподобнее. Для этого необходимо в буфере связанных элементов сохранять, как минимум, нормали объектов и коэффициенты шероховатости. Потом использовать алгоритмы освещения, примерно такие, какие были изложены в п. 4.2 и п. 4.4. И только потом проводить сортировку и смешивание полупрозрачных фрагментов. Подробное рассмотрение вопросов освещения полупрозрачных объектов выходит за рамки настоящего пособия, но автор надеется, что читатель самостоятельно и творчески сможет решить этот вопрос.

4.6. Экранное сглаживание

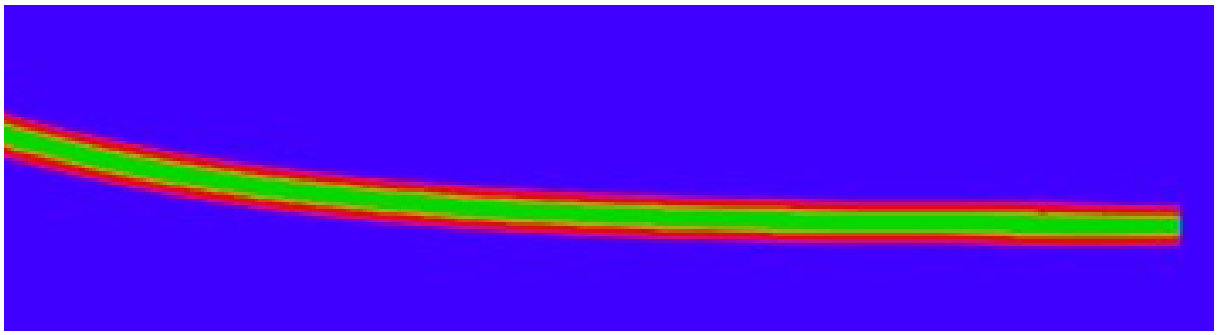
Экранное сглаживание (anti-aliasing) – это технология, которая в общем случае борется с потерей видеоинформации из-за дискретности дисплея или монитора. Например, на рис. 4.24, *а* показана увеличенная часть экрана, на которой видна зубчатость, или ступенчатость, изображения. Это один из самых ярких примеров негативных эффектов, которые возникают из-за конечного числа пикселов на экране монитора. Причем потеря информации может происходить не только

в пространстве монитора, но и во времени, при отображении динамических сцен, что вызывает негативное мерцание или дрожание отдельных частей изображения.

На сегодняшний день существует множество алгоритмов борьбы с подобными негативными эффектами, которые называют *алгоритмами экранного сглаживания*. На рис. 4.24, *б* показан результат работы одного из таких алгоритмов.



а)



б)

Рис. 4.24. Применение алгоритмов сглаживания изображения: а – изображение без сглаживания; б – изображение со сглаживанием

В современных видеокартах аппаратно реализован метод сглаживания MSAA (multi-sample anti-aliasing), основанный на многих выборках (сэмплах) для каждого пиксела и последующем вычислении результата. Современные видеокарты поддерживают от 2 до 16 выборок (MSAA-2, MSAA-4, MSAA8, MSAA16). На рис. 4.25 приведен пример применения алгоритма MSAA, прямые нарисованы в таком порядке: без MSAA, MSAA-2, MSAA-4, MSAA-8. Однако надо учитывать, что увеличение количества выборок требует и большего времени. Кроме того, применение такого алгоритма приводит к ряду проблем, особенно при алгоритмах отложенного освещения.

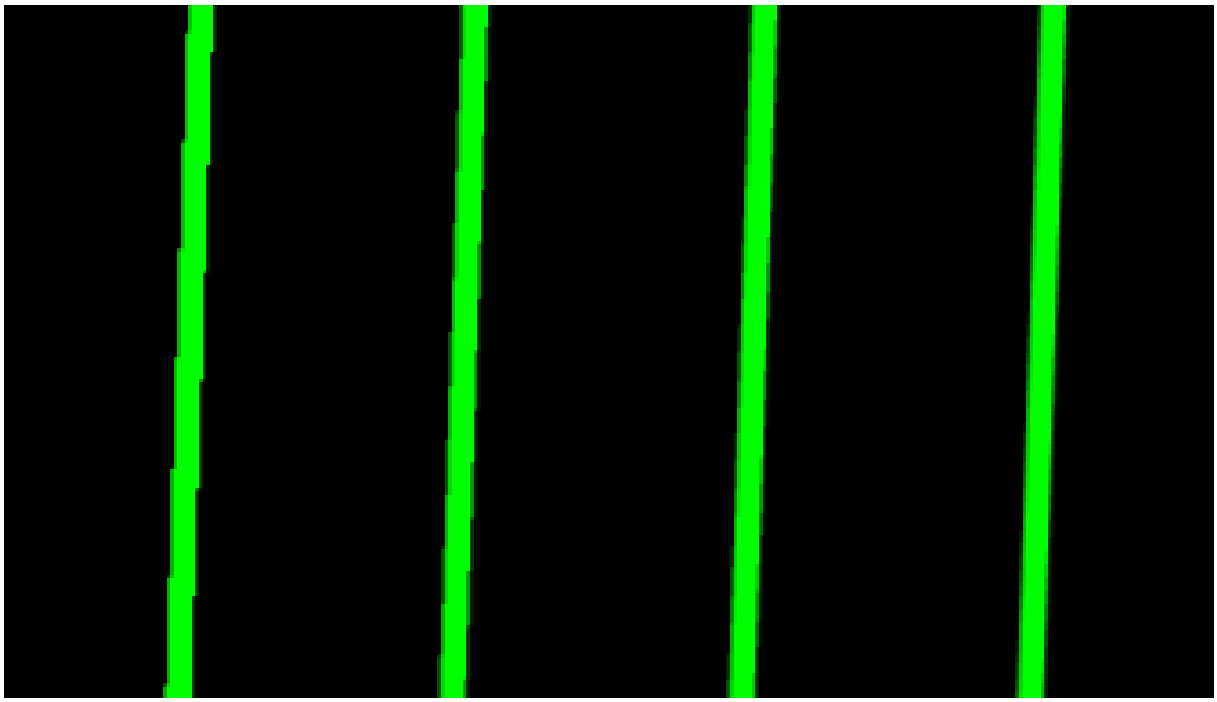


Рис. 4.25. Пример работы алгоритма сглаживания MSAA

Для применения аппаратного сглаживания MSAA необходимо при создании текстур, или цепочки отображения, или поверхности рендеринга в поле SampleDesc структуры, которая описывает создаваемый ресурс, установить следующие значения:

swapChainDesc.SampleDesc.Count = 8;

swapChainDesc.SampleDesc.Quality =

D3D11_STANDARD_MULTISAMPLE_PATTERN;

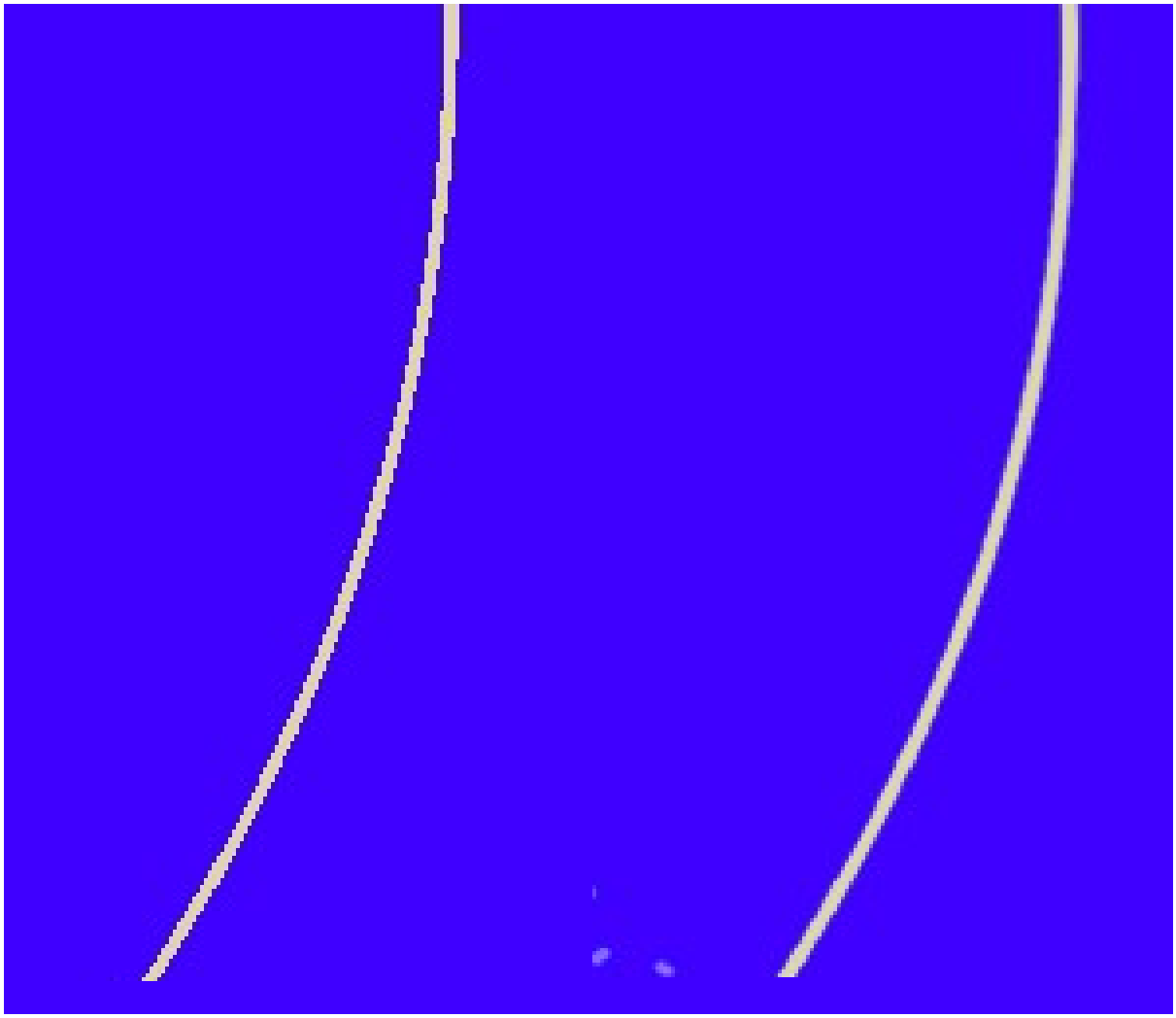
А в структуре D3D11_RASTERIZER_DESC, которая отвечает за состояние растеризации подтвердить использование мультисэмплирования

desc.MultisampleEnable=true;

В качестве альтернативы можно применять иные алгоритмы сглаживания [12]. Например, основанные на аналитических методах (алгоритмы GBAA, SRAA и др.) или на финальной фильтрации (алгоритмы MLAA, FXAA, SMAA и др.). Наиболее популярные на сегодняшний день – алгоритмы временной, или темпоральной, фильтрации TAA. Тем не менее универсального решения не существует и многие разработчики используют комплексные подходы.

В этом параграфе рассмотрим **алгоритм постпроцессорной фильтрации FXAA** (Fast approXimate Anti-Aliasing). Алгоритм FXAA представляет собой пиксельный шейдер, который применяется ко

всей рабочей области окна отображения на финальной стадии. Алгоритм FXAA более производительный, чем MSAA, однако за счет некоторой потери качества. На рис. 4.24 и 4.26, *а, б* приведены примеры применения этого алгоритма.

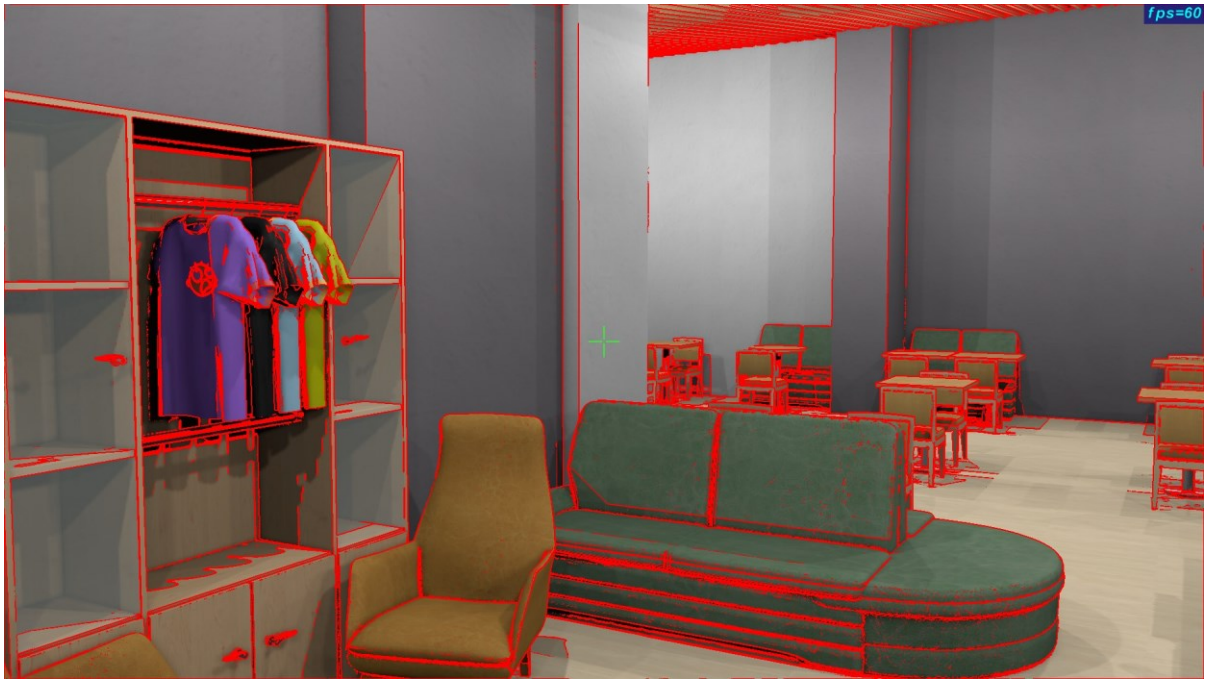


а) *б)*

Рис. 4.26. Пример сглаживания алгоритмом FXAA

Алгоритм FXAA [13; 14] основан на получении скалярной оценки яркости каждого пиксела, нахождении неоднородностей и считывании из входного изображения так называемых субпикселей, с дальнейшей фильтрацией. На рис. 4.27 показаны выделенные красным пикселы, которые, по мнению алгоритма FXAA, требуют сглаживания. Как видно из рисунка, это в большинстве своем пикселы на границах объектов или переходов между тенью и светом.

а)



б)

Рис. 4.27. Вычисление перепадов яркости алгоритмом FXAA

Поскольку входное изображение является, по сути, текстурой, то, считывая пикселы из этой текстуры (текселы) с некоторым сдвигом менее чем на тексел, можно сразу получить аппаратную фильтрацию, которая обязательно должна быть установлена в настройках

фильтрации текстур. Пиксельный шейдер, реализующий алгоритм FXAA, приведен в публикациях [15; 16] и в прил. 10.

Следует обратить внимание на функцию этого пиксельного шейдера toGamma(), которая реализует еще одну финальную обработку построенного трехмерного изображения, а именно гамма-коррекцию. Необходимость гамма-коррекции отмечалась ранее в п. 3.4 в связи с тем, что человеческий глаз нелинейно воспринимает яркость. Ниже приведены два варианта кода функции гамм-коррекции

```
float3 toGamma(float3 lin)
{ lin = max(6.10352e-5, lin);
return min(lin * 12.92, pow(max(lin, 0.00313067), 1.0 / 2.4) * 1.055
- 0.055); }
float3 toGamma(float3 lin)
{ return pow(lin,1.0/2.2);}
```

Контрольные вопросы

1. В чем заключается метод отложенного освещения?
2. Какие существенные плюсы дает метод отложенного освещения?
3. Каковы основные недостатки метода отложенного освещения?
4. Что такое G-буфер?
5. Какие данные должен содержать G-буфер?
6. Как перевести координаты точки из экранного пространства в мировое пространство?
7. Приведите алгоритм освещения точечным источником с применением G-буфера.
8. Что такое буфер трафарета?
9. Как установить параметры буфера трафарета?
10. Приведите основные формулы для вычисления коэффициента освещения конусного источника света.
11. Каков основной недостаток алгоритмов освещения на основе G-буфера?
12. Что такое затенение окружением?
13. Какие основные параметры в методе затенения окружением?
14. Каков основной алгоритм формирования карты теней?

15. Какой тип текстуры применяют в качестве карты теней для точечного источника освещения?

16. Какие основные недостатки содержит метод построения карты теней?

17. В чем заключается негативный эффект «Питера Пена»?

18. Какой фильтр используют для сглаживания «ступенчатости» теней, полученных с использованием теневых карт?

19. Как подключить аппаратную фильтрацию ступенчатости теневых карт?

20. В чем сложность отображения полупрозрачных объектов?

21. Каковы основные алгоритмы сортировки полупрозрачных объектов?

22. В чем заключается алгоритм OIT?

ЗАКЛЮЧЕНИЕ

В учебном пособии были рассмотрены основные понятия и определения трехмерной графики, дано описание вершин и их атрибутов, матричных преобразований и буфера глубины. Приведена схема графического конвейера, рассмотрены его основные этапы, стадии управления и программирования.

Изложены средства инициализации и показаны возможности библиотеки DirectX 11. Дано понятие вершинных, индексных и константных буферов. Показаны инструменты библиотеки DirectX для управления стадиями графического конвейера. Описаны программируемые стадии графического конвейера: вершинные, фрагментные и геометрические шейдеры. Приведены примеры отображения трехмерных объектов, их текстурирования и полупрозрачного смешивания.

Особое внимание в пособии уделено вопросам освещения и отображения трехмерных сцен. Рассмотрены различные типы источников света, модели затенения по Гуро, Фонгу и Куку – Торренсу. Дано понятие физически корректного рендеринга. Показаны примеры использования карт нормалей и карт окружения. Подробно разобран алгоритм отложенного освещения. Приведены примеры применения множества источников освещения в трехмерных сценах, методов затенения окружением и использования кубических теневых карт. Приведены алгоритмы отображения множества полупрозрачных объектов в трехмерных сценах, а также проведен обзор алгоритмов экранного сглаживания и подробно рассмотрен алоритм FXAA.

В приложениях к учебному пособию приведены некоторые реализации упомянутых ранее алгоритмов и шейдеров. А на сайте https://github.com/samoyow1973 прдставлен ряд примеров программ к некоторым главам и параграфам. Учебное пособие направлено в первую очередь на аудиторию начинающих программистов трехмерной графики и содержит только основное и далеко не полное описание как методов, интерфейсов и возможностей библиотеки DirectX 11, так и алгоритмов визуализации трехмерных сцен. Тем не менее автор выражает надежду, что настоящее учебное пособие позволит читателю открыть двери в увлекательный мир программирования трехмерной графики.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Ericson, C. Real-Time Collision Detection / C. Ericson. – San Francisco : Elsevier Inc., 2005. – 591 p. – ISBN 1-55860-732-3.

2. Sulaiman, H. A. Bounding Volume Hierarchies for Collision Detection [Электронный ресурс] / H. A. Sulaiman, A. Bade. – URL: https://www.semanticscholar.org/reader/eb239bad1249db0cc57dee2cca4623c436a01c58 (дата обращения: 25.11.2024).

3. Eimandar, P. DirectX 11. Game programing / P. Eimandar. – Birmingham : Packt Publishing, 2013. – 146 p. – ISBN 978-1849694803.

4. Luna, F. Introduction to 3D Game Programming with DirectX 11 / F. Luna. – Herndon : Mercury Learning and Information, 2012. – 600 p. – ISBN 978-1936420223.

5. Chubak, B. Немного о затенении по Фонгу [Электронный ресурс] / B. Chubak. – URL: https://habr.com/ru/articles/441862/ (дата обращения: 25.11.2024).

6. Кулагин, Д. А. Модели затенения [Электронный ресурс] / Д. А. Кулагин. – URL: https://compgraphics.info/3D/lighting/shading_model.php (дата обращения: 25.11.2024).

7. Бусаров, А. Готовим Phisicaly Based rendering + Image-based Lighting. Теория+практика. Шаг за шагом [Электронный ресурс] / А. Бусаров. – URL: https://habr.com/ru/articles/326852/ (дата обращения: 25.11.2024).

8. DeVries, J. Learn OpenGL. Урок 6.1. PBR или Физически-корректный рендеринг. Теория [Электронный ресурс] / J. DeVries. – URL: https://habr.com/ru/articles/426123/ (дата обращения: 25.11.2024).

9. Mendez, J. M. A Simple and Practical Approach to SSAO [Электронный ресурс] / J. M. Mendez. – URL: https://www.gamedev.net/tutorials/_/technical/graphics-programming-and-theory/a-simple-and-practical-approach-to-ssao-r2753/ (дата обращения: 12.10.2024).

10. Нативная реализация OmniDirectional теней в DirectX11 [Электронный ресурс]. – URL: https://habr.com/ru/articles/259679/ (дата обращения 12.10.2024).

11. Кузнецов, Р. Алгоритм Order-Independent Transparency с использованием связных списков на Direct3D 11 и OpenGL 4 [Электронный ресурс] / Р. Кузнецов. – URL: https://habr.com/ru/articles/224003/ (дата обращения: 12.10.2024).

12. Depth Peel или разложение сцены по слоям [Электронный ресурс]. – URL: https://steps3d.narod.ru/tutorials/depth-peel-tutorial.html (дата обращения: 22.09.2024).

13. OIT - Order Independent Transparency [Электронный ресурс]. – URL: https://orenk2k.blogspot.com/2010/10/oit-order-independent-transparency-3.html (дата обращения: 22.09.2024).

14. Bavoil, L. Order Independent Transparency with Dual Depth Peeling [Электронный ресурс] / L. Bavoil, K. Myers. – URL: https://developer.download.nvidia.com/SDK/10.5/opengl/src/dual_depth_peeling/doc/DualDepthPeeling.pdf (дата обращения: 01.10.2024).

15. Введение в дискретно-ориентированные многогранники для задачи определения столкновений. [Электронный ресурс]. – URL: https://habr.com/ru/articles/257339/ (дата обращения: 18.10.2024).

16. Тужик, А. Всё, что вы хотели знать про АнтиАлиасинг. [Электронный ресурс] / А. Тужик. – URL: https://gamedev.ru/code/articles/aa#ssaa (дата обращения: 28.11.2024).

17. Lottes, T. FXAA [Электронный ресурс] / T. Lottes. – URL: https://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf (дата обращения: 16.11.2024).

18. Atwood, J. Fast Approximate Anti-Aliasing (FXAA) [Электронный ресурс] / J. Atwood. – URL: https://blog.codinghorror.com/fast-approximate-anti-aliasing-fxaa/ (дата обращения: 09.11.2024).

19. Rodriguez, S. Implementing FXAA [Электронный ресурс] / S. Rodriguez. – URL: https://blog.simonrodriguez.fr/articles/2016/07/implementing_fxaa.html (дата обращения: 11.10.2024).

20. Rodriguez, S. NVIDIA FXAA 3.11 [Электронный ресурс] / S. Rodriguez. – URL: https://gist.github.com/kosua20/0c506b81b3812ac900048059d2383126 (дата обращения: 14.10.2024).

21. Teschner, M. Collision Handling in Dynamic Simulation Environments [Электронный ресурс] / M. Teschner, B. Heidelberger, D. Manocha. – URL: https://www.researchgate.net/publication/245689995_Collision_Handling_in_Dynamic_Simulation_Environments (дата обращения: 22.09.2024).

22. Franke, T. A. Notes on importance sampling [Электронный ресурс] / T. A. Franke. – URL: blog.tobias-franke.eu/2014/03/30/notes_on_importance_sampling.html (дата обращения: 13.09.2024).

23. Heitz, E. Understanding the Masking-Shadowing Function in Microfacet-Based BRDFs [Электронный ресурс] / E. Heitz. – URL: hal.inria.fr/hal-00942452v1/document (дата обращения: 22.10.2024).

24. Microfacet Models for Refraction through Rough Surfaces [Электронный ресурс] / B. Walter [и др.] // Eurographics Symposium on Rendering, 2007. – URL: www.cs.cornell.edu/~srm/publications/EGSR07-btdf.pdf (дата обращения: 22.10.2024).

25. Karis, B. Real Shading in Unreal Engine [Электронный ресурс] / B. Karis. – URL: de45xmedrsdbp.cloudfront.net/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf (дата обращения: 22.10.2024).

26. Stevens, J. Physically Based Rendering Algorithms: A Comprehensive Study In Unity3D [Электронный ресурс] / J. Stevens. – URL: www.jordanstevenstechart.com/physically-based-rendering (дата обращения: 25.11.2024).

27. Hammon, E. PBR Diffuse Lighting for GGX+ Smith Microsurfaces [Электронный ресурс] / E. Hammon // Smith Microsurfaces. – URL: https://ubm-twvideo01.s3.amazonaws.com/o1/vault/gdc2017/Presentations/Hammon_Earl_PBR_Diffuse_Lighting.pdf (дата обращения: 25.11.2024).

28. Справочник по HLSL [Электронный ресурс]. – URL: https://learn.microsoft.com/ru-ru/windows/win32/direct3dhlsl/dx-graphics-hlsl-reference (дата обращения: 28.11.2024).

29. Vivo, P. G. The book of shaders [Электронный ресурс] / P. G. Vivo, J. Lowe. – URL: https://thebookofshaders.com/ (дата обращения: 28.11.2024).

30. Alamia, M. Physically Based Rendering – Cook-Torrance [Электронный ресурс] / M. Alamia. – URL: http://www.codinglabs.net/article_physically_based_rendering_cook_torrance.aspx (дата обращения: 19.10.2024).

31. DirectX 11 шаг за шагом [Электронный ресурс] // Fandom. – URL: https://directx.fandom.com/ru/wiki/DirectX_11_шаг_за_шагом (дата обращения: 07.10.2024).

ПРИЛОЖЕНИЯ

Приложение 1

Функции инициализации и основного цикла отображения 3D-сцены

```
#pragma comment(lib, “d3d11.lib”)
#pragma comment(lib, «dxgi.lib”)
#pragma comment(lib, “D3Dcompiler.lib”)
#pragma comment(lib, “DirectXTK.lib”)
#pragma comment(lib, “DirectXTex.lib”)
#include <windows.h>
#include <windowsx.h>
#include <directxmath.h>
#include <d3d11.h>
using namespace DirectX;
class D3D
{
public:
    // Интерфейс для управления рендерингом
    IDXGISwapChain* swapChain=NULL;
    ID3D11Device* device=NULL;
    ID3D11DeviceContext* deviceContext=NULL;
    // Указатели на поверхности рендеринга, буфера
глубины и текстуры с буфером глубины
    ID3D11RenderTargetView* target=NULL;
    ID3D11DepthStencilView* depth_stencil = NULL;
    ID3D11ShaderResourceView* depthStencilTexture =
NULL;

    INT videoCardMemory; // размер памяти видеокарты
    WCHAR videoCardDescription[128]; // название ви-
деокарты
```

```
    BOOL vsync_enabled=TRUE; // вертикальная синхрони-
зация
    BOOL fullscreen = FALSE;
    UINT winwidth=0; // Размер окна вывода
    UINT winheight=0;
    UINT screenwidth=0; // Размер экрана
    UINT screenheight=0;

public:
    // Функция инициализации
    HRESULT Initialize(int, int, bool, HWND, bool);
    // Функция начала отрисовки 3D-сцены
    VOID BeginScene(XMFLOAT4*);

    // Функция для представления результата с готовой
3D-сценой
    VOID EndScene();
    D3D(); // конструктор
    ~D3D();// деструктор
};
D3D::D3D()
{
    this->device = NULL;
    this->deviceContext = NULL;
    this->swapChain = NULL;
}

D3D::~D3D()
{
    // Выключить полноэкранный режим
    if (this->swapChain != NULL) swapChain-
>SetFullscreenState(false, NULL);
    // Удалить все интерфейсы
    ReleaseCOM(this->deviceContext);
```

```
    ReleaseCOM(this->device);
    ReleaseCOM(this->swapChain);
    ReleaseCOM(this->target);
    ReleaseCOM(this->depthStencilTexture);
    ReleaseCOM(this->depth_stencil);

}
HRESULT D3D::Initialize(int winwidth, int winheight,
bool vsync, HWND hwnd, bool fullscreen)
{
    // Запомним параметры окна вывода, экрана и режима
вывода
    this->vsync_enabled = vsync;
    this->winwidth = winwidth;
    this->winheight = winheight;
    this->screenwidth = GetSystemMetrics(SM_CXSCREEN);
    this->screenheight=GetSystemMetrics(SM_CYSCREEN);
    this->fullscreen = fullscreen;
    CoInitializeEx(nullptr, COINIT_MULTITHREADED);
    IDXGIFactory* factory;
    IDXGIAdapter* adapter;
    IDXGIOutput* adapterOutput;
    unsigned int numModes, i, numerator, denominator;
    numModes = 0;
    DXGI_MODE_DESC* displayModeList;
    DXGI_ADAPTER_DESC adapterDesc;
    int error;
    DXGI_SWAP_CHAIN_DESC swapChainDesc;
    D3D_FEATURE_LEVEL featureLevel;
    D3D11_VIEWPORT viewport;
    ID3D11Texture2D* pointer = NULL;
    D3D11_TEXTURE2D_DESC depthBufferDesc;
    D3D11_DEPTH_STENCIL_VIEW_DESC depthStencilViewD-
esc;
```

```
    D3D11_SHADER_RESOURCE_VIEW_DESC descSRV;
    // Создание фабрики графических интерфейсов.
    If
(FAILED(CreateDXGIFactory(__uuidof(IDXGIFactory),
(void**)&factory))) return E_FAIL;
    // Создание интерфейса основного графического
адаптера (видеокарты).
    If (FAILED(factory->EnumAdapters(0, &adapter)))
return E_FAIL;
    // Создание интерфейса основного монитора.
    If (FAILED(adapter->EnumOutputs(0,
&adapterOutput))) return E_FAIL;
    // Число режимов с форматом
DXGI_FORMAT_R8G8B8A8_UNORM
    if (FAILED(adapterOutput-
>GetDisplayModeList(DXGI_FORMAT_R8G8B8A8_UNORM,
DXGI_ENUM_MODES_INTERLACED, &numModes, NULL))) return
E_FAIL;
    // Создание массива режимов
    displayModeList = new DXGI_MODE_DESC[numModes];
    if (!displayModeList) return E_FAIL;
    // Заполнение массива режимами с форматом
DXGI_FORMAT_R8G8B8A8_UNORM.
    If (FAILED(adapterOutput-
>GetDisplayModeList(DXGI_FORMAT_R8G8B8A8_UNORM,
DXGI_ENUM_MODES_INTERLACED, &numModes, displayModeL-
ist))) return E_FAIL;
    // Находим режим, совпадающий с текущим размером
экрана
    // И вычисляем частоту обновления кадров в мони-
торе в параметрах numerator and denominator.
    For (i = 0; i < numModes; i++)
    {
```

```
        if (displayModeList[i].Width == (unsigned
int)GetSystemMetrics(SM_CXSCREEN))
        {
            if (displayModeList[i].Height == (unsigned
int)GetSystemMetrics(SM_CYSCREEN))
            {
                numerator = displayModeL-
ist[i].RefreshRate.Numerator;
                denominator = displayModeL-
ist[i].RefreshRate.Denominator;
            }
        }
    }
    // Получаем описание видеокарты.
    If (FAILED(adapter->GetDesc(&adapterDesc)))return
E_FAIL;
    // Вычисляем ее размер в Мб.
    This->videoCardMemory =
(int)(adapterDesc.DedicatedVideoMemory / 1024 / 1024);
    // Запоминаем название видеокарты.
    Error = wcscpy_s(this->videoCardDescription,
adapterDesc.Description);
    if (error != 0) return E_FAIL;
    // Удаляем массив режимов.
    Delete[] displayModeList;
    displayModeList = NULL;
    // Удаляем ненужные интерфейсы.
    ReleaseCOM(adapterOutput);
    ReleaseCOM(adapter);
    ReleaseCOM(factory);
    // Инициализируем структур для цепочки переключе-
ний буферов вывода.
    ZeroMemory(&swapChainDesc, sizeof(swapChainDesc));
    // Устанавливаем 1 задний буфер.
```

```
    swapChainDesc.BufferCount = 1;
    // Устанавливаем размер буфера по размеру окна вы-
вода.
    swapChainDesc.BufferDesc.Width = winwidth;
    swapChainDesc.BufferDesc.Height = winheight;
    // Устанавливаем формат поверхности вывода.
    swapChainDesc.BufferDesc.Format =
DXGI_FORMAT_R8G8B8A8_UNORM;
    // Устанавливаем частоту обновления.
    If (vsync)
    {
        swapChainDesc.BufferDesc.RefreshRate.Numerator
= numerator;
        swapChain-
Desc.BufferDesc.RefreshRate.Denominator = denominator;
    }
    else
    {
        swapChainDesc.BufferDesc.RefreshRate.Numerator
= 0;
        swapChain-
Desc.BufferDesc.RefreshRate.Denominator = 1;
    }
    // Устанавливаем тип создаваемой поверхности.
    swapChainDesc.BufferUsage =
DXGI_USAGE_RENDER_TARGET_OUTPUT;
    // Устанавливаем дескриптор окна, в которое будет
производиться вывод.
    swapChainDesc.OutputWindow = hwnd;
    // Выключим мультисемплинг.
    swapChainDesc.SampleDesc.Count = 1;
    swapChainDesc.SampleDesc.Quality = 0;
    // Устанавливаем полноэкранный/оконный режим.
    If (fullscreen) swapChainDesc.Windowed = false;
```

```
    else swapChainDesc.Windowed = true;
    // Масштабные коэффициенты.
    swapChainDesc.BufferDesc.ScanlineOrdering =
DXGI_MODE_SCANLINE_ORDER_UNSPECIFIED;
    swapChainDesc.BufferDesc.Scaling =
DXGI_MODE_SCALING_UNSPECIFIED;
    // После отображения сбросить контент заднего бу-
фера.
    swapChainDesc.SwapEffect =
DXGI_SWAP_EFFECT_DISCARD;
    // Дополнительных флагов нет.
    swapChainDesc.Flags = 0;
    // Установить уровень DirectX.
    featureLevel = D3D_FEATURE_LEVEL_11_0;
    // Создать интерфейсы swap chain, Direct3D device,
и Direct3D device context.
    If (FAILED(D3D11CreateDeviceAndSwapChain(NULL,
D3D_DRIVER_TYPE_HARDWARE, NULL, 0, &featureLevel, 1,
    D3D11_SDK_VERSION,      &swapChainDesc,      &this-
>swapChain,      &this->device,      NULL,      &this-
>deviceContext))) return E_FAIL;
    // Получить указатель на задний буфер в памяти.
    If (FAILED(this->swapChain->GetBuffer(0,
__uuidof(ID3D11Texture2D), (LPVOID*)&pointer))) return
E_FAIL;
    // Создать цель рендеринга.
    If (FAILED(this->device-
>CreateRenderTargetView(pointer, NULL, &this-
>target))) return E_FAIL;
    // удалить временный интерфейс
    ReleaseCOM(pointer);
    // Очистить структуру для параметров буфера глуби-
ны.
```

```
    ZeroMemory(&depthBufferDesc,
sizeof(depthBufferDesc));
    // Установить параметры для буфера глубины.
    depthBufferDesc.Width = winwidth;
    depthBufferDesc.Height = winheight;
    depthBufferDesc.MipLevels = 1;
    depthBufferDesc.ArraySize = 1;
    depthBufferDesc.Format =
DXGI_FORMAT_R24G8_TYPELESS;
    depthBufferDesc.SampleDesc.Count = 1;
    depthBufferDesc.SampleDesc.Quality = 0;
    depthBufferDesc.Usage = D3D11_USAGE_DEFAULT;
    depthBufferDesc.BindFlags =
D3D11_BIND_SHADER_RESOURCE | D3D11_BIND_DEPTH_STENCIL;
    depthBufferDesc.CPUAccessFlags = 0;
    depthBufferDesc.MiscFlags = 0;
    // Получить указатель на буфер глубины в памяти
    if (FAILED(this->device-
>CreateTexture2D(&depthBufferDesc, NULL, &pointer)))
return E_FAIL;
    // Очистить и заполнить структуру для текстуры бу-
фера глубины
    ZeroMemory(&descSRV, sizeof(descSRV));
    descSRV.Format =
DXGI_FORMAT_R24_UNORM_X8_TYPELESS;
    descSRV.ViewDimension =
D3D11_SRV_DIMENSION_TEXTURE2D;
    descSRV.Texture2D.MipLevels = 1;
    descSRV.Texture2D.MostDetailedMip = 0;
    // Создать текстуру с буфером глубины
    if (FAILED(this->device-
>CreateShaderResourceView(pointer, &descSRV, &this-
>depthStencilTexture))) return E_FAIL;
```

```
    // Очистить структуру с параметрами цели вывода
буфера глубины.
    ZeroMemory(&depthStencilViewDesc,
sizeof(depthStencilViewDesc));
    // Set up the depth stencil view description.
    depthStencilViewDesc.Format =
DXGI_FORMAT_D24_UNORM_S8_UINT;
    depthStencilViewDesc.ViewDimension =
D3D11_DSV_DIMENSION_TEXTURE2D;
    depthStencilViewDesc.Texture2D.MipSlice = 0;
    // Создать цель вывода буфера глубины.
    If (FAILED(this->device-
>CreateDepthStencilView(pointer,
&depthStencilViewDesc, &this->depth_stencil))) return
E_FAIL;
    // удалить временный интерфейс
    ReleaseCOM(pointer);
    // Установить параметры окна вывода DirectX (вью-
порта).
    Viewport.MinDepth = 0.0f;
    viewport.MaxDepth = 1.0f;
    viewport.Width = (float)winwidth;
    viewport.Height = (float)winheight;
    viewport.TopLeftX = 0.0f;
    viewport.TopLeftY = 0.0f;
    // Создать вьюпорт.
    This->deviceContext->RSSetViewports(1, &viewport);
    // Присоединить цель рендеринга и цель буфера глу-
бины к конвееру рендеринга
    this->deviceContext->OMSetRenderTargets(1, &this-
>target, this->depth_stencil);
    return S_OK;
}
VOID D3D::BeginScene(XMFLOAT4* color)
```

```
{
    // Очистить поверхность рендеринга
    this->deviceContext->ClearRenderTargetView(this-
>target, (FLOAT*)color);
    // Очистить буфер глубины
    this->deviceContext->ClearDepthStencilView(this-
>depth_stencil, D3D11_CLEAR_DEPTH |
D3D11_CLEAR_STENCIL, 1.0f, 0);
return;
}
void D3D::EndScene()
{
    if (this->vsync_enabled) this->swapChain-
>Present(1, 0);// Дождаться вертикальной синхронизации
и отобразить.
    Else this->swapChain->Present(0, 0); // Отобразить
быстро, как только возможно.
    Return;
}
```

Приложение 2

Вершинный и пиксельный шейдеры затенения по Фонгу

Вершинный шейдер:

```
cbuffer CBUFFER0:register (b0)
{
 float4 time0;
 float4x4 vpmatrix;
 float4x4 wmatrix;
 float4 l_color;
 float4 l_pos;
 float4 l_dir;
 float4 l_param;
 float4 view_pos;
};
struct INPUT
{
 float3 position : POSITION;
 float3 normal : NORMAL;
 float4 color : COLOR0;
 float2 tex : TEXCOORD0;
};
struct OUTPUT
{
 float4 position : SV_POSITION;
 float3 normal : NORMAL;
 float2 tex:TEXCOORD0;
 float3 V:TEXCOORD1;
 float3 L:TEXCOORD2;
};
OUTPUT main(INPUT input)
```

```
{
 OUTPUT output;

 float3 vpos= mul(wmatrix, float4(input.position,
1.0)).xyz;
 output.position = mul(vpmatrix, float4(vpos, 1.0));
 output.normal = normalize(mul(wmatrix,
float4(input.normal, 0.0)).xyz);
 output.V = normalize(view_pos.xyz – vpos);
 output.L = normalize(l_pos.xyz – vpos);
// output.L= l_dir.xyz; // для направленного источника
 output.tex = input.tex;
 return output;
}
```

Пиксельный шейдер:

```
struct INPUT
{
 float4 position : SV_POSITION;
 float3 normal : NORMAL;
 float2 tex:TEXCOORD0;
 float3 V:TEXCOORD1;
 float3 L:TEXCOORD2;
};

float4 main(INPUT input) : SV_TARGET
{
 float ambient = 0.1;
 float3 diffuse_albedo = float3(0.5,0.2,0.7);
 float specular_power = 8;
 float3 specular_albedo = float3(0.5, 0.5, 0.5);

 float3 N = normalize(input.normal);
 float3 L = normalize(input.L);
 float3 V = normalize(input.V);
```

```
 float3 diffuse = max(dot(N, L), 0.0) * dif-
fuse_albedo;
 // Затенение по Фонгу
 // float3 R = reflect(L, N);
 // float3 specular = pow(max(dot(R, V), 0.0), specu-
lar_power) * specular_albedo;
 // Затенение Лина – Фонга
 float3 H = normalize(V+L);
 float3 specular = pow(max(dot(H, N), 0.0), specu-
lar_power*4) * specular_albedo;

 return float4(ambient+diffuse+specular,1.0);
}
```

***Приложение* 3**

Шейдер модели освещения Кука – Торренса

Пиксельный шейдер:

```
#define PI 3.1415926
Texture2D Texture: register(t0);
SamplerState SampleType: register(s0);

struct INPUT
{
 float4 position : SV_POSITION;
 float3 normal : NORMAL;
 float2 tex:TEXCOORD0;
 float3 V:TEXCOORD1;
 float3 L:TEXCOORD2;
};
float GGX_G(float cosTheta, float alpha)
{
 float cosTheta_sqr = saturate(cosTheta * cosTheta);
 float tan2 = (1 – cosTheta_sqr) / cosTheta_sqr;
 float GP = 2 / (1 + sqrt(1 + alpha * alpha * tan2));
 return GP;
}
float GGX_D(float cosTheta, float alpha)
{
 float alpha2 = alpha * alpha;
 float NH_sqr = saturate(cosTheta * cosTheta);
 float den = NH_sqr * alpha2 + (1.0 – NH_sqr);
 return alpha2 / (PI * den * den);
}
float3 FresnelSchlick(float3 F0, float cosTheta)
{
```

```
 return F0 + (1.0 - F0) * pow(1.0 - satu-
rate(cosTheta), 5.0);
}

float4 main(INPUT input) : SV_TARGET
{
 float3 ambient = float3(0.004,0.0032,0.0024);
 float3 albedo_0 = float3(0.6,0.48,0.36);
 float3 F0_0 = float3(0.04, 0.04, 0.04);
 float Km = 0.0;
 float roughness = 0.3;

 float3 color_tex=Texture.Sample(SampleType, in-
put.tex).rgb;
 float3 color_a = ambient * color_tex;
 float3 N = normalize(input.normal);
 float3 L = normalize(input.L);
 float3 V = normalize(input.V);
 float3 H = normalize(input.V + input.L);

 float NL = dot(N, L);
 if (NL <= 0.0) return float4(pow(color_a, 1.0 / 2.2),
1.0);
 float NV = dot(N, V);
 if (NV <= 0.0) return float4(pow(color_a, 1.0 / 2.2),
1.0);
 float NH = dot(N, H);
 float HV = dot(V, H);

 float3 F0 = F0_0 * (1.0 - Km) + albedo_0 * Km;
 float3 albedo = albedo_0 * (1.0 - Km) + F0_0 * Km;
```

```
 float roug_sqr = roughness * roughness;
 float G = GGX_G(NV, roug_sqr) * GGX_G(dot(N, L),
roug_sqr);
 float D = GGX_D(dot(N, H), roug_sqr);
 float3 F= FresnelSchlick(F0, HV);

 float3 specK = max(0.0,G * D * F * 0.25 / NV);
 float3 diffK = saturate(1.0 – F)* albedo* NL / PI;
 float3 color = diffK * color_tex + specK + color_a;
 color = pow(color, 1.0 / 2.2);
 return float4(color,1.0);
}
```

Приложение 4

Восстановление позиции точки в мировом пространстве из экранного пространства

```
cbuffer Cbuffer:register (b0)
{
float4 resolution; // Размеры окна в пикселах
float4x4 inv_veiw_proj; // Инвертированная матрица
произведения матриц вида и проекции
}
struct PS_INPUT
{
    float4 position : SV_POSITION;
};
float3 WSPosition(float3 pos)
{
    // Получить x/w и y/w в экранном пространстве
    float x = pos.x * 2 – 1;
    float y = (1 – pos.y) * 2 – 1;
    float4 vProjectedPos = float4(x, y, z, 1.0f);
    // трансформировать обратным преобразованием
 // view_proj матрицы
    float4 vPositionVS = mul(inv_veiw_proj, vProject-
edPos);
    // Делить на w
    return   vPositionVS.xyz / vPositionVS.w;
}
float4 main(PS_INPUT input) :SV_TARGET
{
float3 pos = input.position.xyz;
pos.xy/= resolution.xy; // перевести в относительные
                        // значения
// Восстановление координат в мировом пространстве
float3 w_pos = WSPosition(pos);
}
```

Приложение 5

Расчет векторов в пространстве TBN

```
VOID CalculateTBN(VERTEX* v0, VERTEX* v1, VERTEX* v2,
XMFLOAT3* normal, XMFLOAT3* tangent, XMFLOAT3* binor-
mal)
{
      XMVECTOR vector1, vector2;
      XMVECTOR tuv[2];
      FLOAT den;
      // Рассчитать два вектора в плоскости треугольника.
      Vector1 = XMLoadFloat3(&v1->pos) - XMLoad-
Float3(&v0->pos);
      vector2 = XMLoadFloat3(&v2->pos) - XMLoad-
Float3(&v0->pos);
      // Рассчитать два вектора в координатах текстур
треугольника.
      Tuv[0] = XMLoadFloat2(&v1->tex) - XMLoad-
Float2(&v0->tex);
      tuv[1] = XMLoadFloat2(&v2->tex) - XMLoad-
Float2(&v0->tex);
      // Вычислить знаменатель бинормального уравнения.
      Den = 1.0f / (XMVectorGetX(tuv[0]) * XMVec-
torGetY(tuv[1]) - XMVectorGetX(tuv[1]) * XMVec-
torGetY(tuv[0]));
      // Вычислить векторное произведение и умножить
на коэффициент
      XMVECTOR tangent_v = XMVec-
tor3Normalize((XMVectorGetY(tuv[1]) * vector1 - XMVec-
torGetY(tuv[0]) * vector2) * den);
      XMVECTOR binormal_v = XMVec-
tor3Normalize((XMVectorGetX(tuv[0]) * vector2 - XMVec-
torGetX(tuv[1]) * vector1) * den);
```

```
    if (tangent != NULL) XMStoreFloat3(tangent, tan-
gent_v);
    if (binormal != NULL) XMStoreFloat3(binormal, bi-
normal_v);
   // нормаль – это векторное произведение тангенты и
бинормали
    XMVECTOR normal_v = XMVec-
tor3Normalize(XMVector3Cross(tangent_v, binormal_v));
   if (normal != NULL) XMStoreFloat3(normal, nor-
mal_v);}
```

Приложение 6

Пиксельный шейдер расчета затенения окружением (SSAO)

```
Texture2D Normal: register(t1);
Texture2D Depth: register(t3);
Texture2D Random: register(t7);

SamplerState Sample1: register(s1);

cbuffer Null:register (b0)
{
    float4 resolution;
float4 plans; // x-near,y-far
}
struct PS_INPUT
{
    float4 position : SV_POSITION;
    float2 tuv : TEXCOORD0;
};

float3 WSPosition(float z, float2 tex_coord)
{
    float x = tex_coord.x * 2 – 1;
    float y = (1 – tex_coord.y) * 2 – 1;
    float4 vProjectedPos = float4(x, y, z, 1.0f);
    float4 vPositionVS = mul(inv_veiw_proj, vProject-
edPos);
    return   vPositionVS.xyz / vPositionVS.w;
}
struct PS_OUTPUT
{
    vector diffuse  : COLOR0;
};
```

```
float4 GetPosition(in float2 tuv)
{
    float2 uv = saturate(tuv);
    float z= Depth.Sample(Sample1, uv).r;
    return float4(WSPosition(z, uv), z);
}

float2 GetRandom(in float2 uv)
{
    return normalize(Random.Sample(Sample1, uv /
float2(64.0, 64.0) * resolution.zw).xy * 2.0f - 1.0f);
}

float AmbientOcclusion(in float2 tcoord, in float3 p,
in float3 cnorm,in float4 ssao )
{
    float3 diff = GetPosition(tcoord).xyz - p;
    const float3 v = normalize(diff);

    const float d = length(diff)*ssao.z;
    return max(0.0, dot(cnorm, v) -
ssao.y)/(1.0+d*d)*ssao.w;
}

float4 main(PS_INPUT input) :SV_TARGET
{
float4 ssao=float4(0.05,0.05,5.0,1.0);
    float2 pos = input.tuv;
const float2 vec[4] = { float2(1.0,0.0),float2(-
1.0,0.0),
float2(0.0,1.0),float2(0.0,-1.0) };
    float near = plans.x;
    float far = plans.y;
        float4 normal_texture = Normal.Sample(Sample1,
pos);
    float3 n = normalize(normal_texture.xyz);
```

```
    uint m_index = uint(normal_texture.w * 1023.0 +
0.5);
    MAT mat = material[m_index];

    float4 ssao = mat.ssao;
    // Позиция
    float4 p = GetPosition(pos); //
    float2 rand = GetRandom(pos);
    float ao = 0.0f;
    // Линеаризовать глубину для диапазона от 0 до 1
    float z =p.w * 2.0 - 1.0;
    float depth= (2.0 * near * far) / (far + near - z
* (far - near));
    depth = (depth - near) / (far - near);
    float rad = 0.01*ssao.x /depth;
    int iterations = 4;
    for (int j = 0; j < iterations; ++j)
    {
        float2 coord1 = reflect(vec[j], rand)*rad;
        float2 coord2 = float2(coord1.x*0.707 -
coord1.y*0.707,
coord1.x*0.707 + coord1.y*0.707);
        ao += AmbientOcclusion(pos + coord1*0.25,
p.xyz, n,ssao);
        ao += AmbientOcclusion(pos + coord2*0.5,
p.xyz, n,ssao);
        ao += AmbientOcclusion(pos + coord1*0.75,
p.xyz, n,ssao);
        ao += AmbientOcclusion(pos + coord2, p.xyz,
n,ssao);
    }
    ao /= (float)iterations*4.0;
    ao = max(0,1.0 - ao);
    return float4(ao, ao, ao, 1.0);
}
```

Приложение 7

Шейдеры для генерации карты теней

Вершинный шейдер:

```
cbuffer WORLD:register (b2) // мировая матрица
{
 float4x4 wmatrix;
};
struct VertexInputType
{
 float4 position : POSITION;
};
struct OUTPUT
{
 float4 position : SV_POSITION;
};
OUTPUT main(VertexInputType input)
{
 OUTPUT output;

 input.position.w = 1.0f;
 output.position = mul(wmatrix, input.position);
 return output;
}
```

Геометрический шейдер:

```
cbuffer VIEW:register (b1) // вью-прожект матрица
{
    float4x4 vpmatrix[6];
};
struct GS_INPUT
{
    float4 position: SV_POSITION;
};
```

```
struct GSOutput
{
    float4 position: SV_POSITION;
    uint RTIndex : SV_RenderTargetArrayIndex;
};

[maxvertexcount(18)]
void main(triangle GS_INPUT input[3], inout Tri-
angleStream<GSOutput> CubeMapStream)
{
    [unroll]
    for (int f = 0; f < 6; ++f)
    {
        {
            GSOutput output = (GSOutput)0;

            output.RTIndex = f;

            [unroll]
            for (int v = 0; v < 3; ++v)
            {
                float4 worldPosition = in-
put[v].position;

                output.position = mul(vpmatrix[f],
worldPosition);

                CubeMapStream.Append(output);
            }
            CubeMapStream.RestartStrip();
        }
    }
}
```

Пиксельный шейдер:

```
cbuffer Light:register (b3) // источник света
{
    float4 position;
    float4 direction;
    float4 lambient;
    float4 ldiffuse;
    float4 lspecular;
    float4 attenuation;
    float4 type;
    float4 shadow;
};
struct PS_INPUT
{
    float4 position : SV_POSITION;
};
struct PS_OUTPUT
{
    vector depth   : SV_TARGET0;
};
PS_OUTPUT main(PS_INPUT input)
{
    PS_OUTPUT output;
    float near = shadow.x;
    float far = shadow.y;
    float z0 = input.position.z;
    float z = z0 * 2.0 - 1.0;
    float depth = (2.0 * near * far) / (far + near - z
* (far - near));
    float zd = depth;
    zd /= attenuation.z; // делим на радиус действия
источника света
    output.depth = zd;
    return output;
                                                    }
```

Приложение 8

Примеры функций фильтров PCF в пиксельном шейдере

Фильтр PCF-5

```
SamplerComparisonState Sample2: register(s…);
TextureCube<float> Shadow: register(t…);
static const float2 DiscSamples5[] =
{
// 5 случайных точек по окружности радиусом 2.5
    float2(0.000000, 2.500000),
    float2(2.377641, 0.772542),
    float2(1.469463, -2.022543),
    float2(-1.469463, -2.022542),
    float2(-2.377641, 0.772543),
};
float SampleCubeShadowPCFDisc5(float3 L, float sD)
{
    float3 SideVector = normalize(cross(L, float3(0,
0, 1)));
    float3 UpVector = cross(SideVector, L);
// shadow_resolution-размер карты теней в пикселах
    SideVector /= shadow_resolution;
    UpVector /= shadow_resolution;
    float totalShadow = 0;
    [unroll] for (int i = 0; i < 5; ++i)
    {
        float3 SamplePos = L + SideVector * DiscSam-
ples5[i].x + UpVector *
 DiscSamples5[i].y;
        totalShadow += Shad-
ow.SampleCmpLevelZero(Sample2,SamplePos,sD);
    }
    totalShadow /= 5;
    return totalShadow;
}
```

Фильтр PCF 3x3

```
float PCF(float3 forward, float sD)
{
    float3 right = float3(forward.z, -forward.x, for-
ward.y);
    right -= forward * dot(right, forward);
    right = normalize(right);
    float3 up = cross(right, forward);
    // радиус выборки (shadow_r) домножаем на разреше-
ние карты теней
    float pixel_step = shadow_r*shadow_resolution;
    right *= pixel_step;
    up *= pixel_step;
    float3 v0;
    v0.x = Shadow.SampleCmpLevelZero(Sample2, forward
- right - up, sD).r;
    v0.y = Shadow.SampleCmpLevelZero(Sample2, forward
- up, sD).r;
    v0.z = Shadow.SampleCmpLevelZero(Sample2, forward
+ right - up, sD).r;
    float3 v1;
    v1.x = Shadow.SampleCmpLevelZero(Sample2, forward
- right, sD).r;
    v1.y = Shadow.SampleCmpLevelZero(Sample2, forward,
sD).r;
    v1.z = Shadow.SampleCmpLevelZero(Sample2, forward
+ right, sD).r;
    float3 v2;
    v2.x = Shadow.SampleCmpLevelZero(Sample2, forward
- right + up, sD).r;
    v2.y = Shadow.SampleCmpLevelZero(Sample2, forward
+ up, sD).r;
    v2.z = Shadow.SampleCmpLevelZero(Sample2, forward
+ right + up, sD).r;
    return dot(v0 + v1 + v2, .1111111f);
}
```

Приложение 9

Пиксельные шейдеры дл алгоритма OIT на связанных списках

Шейдер для заполнения связанного списка:

```
Texture2D shaderTexture: register(t0);
SamplerState Sample0: register(s0);
struct PS_INPUT
{
    float4 position: SV_POSITION;
    float2 tuv: TEXCOORD0;
};
struct PS_OUTPUT
{
    vector diffuse  : SV_TARGET0;
};
struct ListNode
{
    uint packedColor;
    uint depth;
    uint next;
};
globallycoherent RWTexture2D<uint> headBuffer;
globallycoherent RWStructuredBuffer<ListNode> frag-
mentsList;
uint packColor(float4 color)
{
    return (uint(color.r * 255) << 24) | (uint(color.g
* 255) << 16) | (uint(color.b * 255) << 8) |
uint(color.a * 255);
}
[earlydepthstencil]
PS_OUTPUT main(PS_INPUT input)
{
```

```
    PS_OUTPUT output;
    output.diffuse = float4(0, 0, 0, 0);
    uint newHeadBufferValue = frag-
mentsList.IncrementCounter();
    if (newHeadBufferValue == 0xffffffff)
    {
    return output;
    }
    float4 color_tex = shaderTexture.Sample(Sample0,
input.tuv);
    uint2 upos = uint2(input.position.xy);
    uint previosHeadBufferValue;
    InterlockedExchange(headBuffer[upos],
newHeadBufferValue, previosHeadBufferValue);
    uint currentDepth = f32tof16(input.position.z);
    ListNode node;
    node.packedColor = packCol-
or(float4(mat.diffuse.rgb * color_tex.rgb * compo-
nent_light.w, mat.diffuse.a* color_tex.a));
    node.depthAndMatterial = currentDepth;
    node.next = previosHeadBufferValue;
    fragmentsList[newHeadBufferValue] = node;
    return output;
}
```

Шейдер для отображения полупрозрачных объектов:

```
static const int MAX_FRAGMENTS = 16;
struct PS_INPUT
{
    float4 position: SV_POSITION;
};
struct PS_OUTPUT
{
    vector diffuse  : SV_TARGET0;
};
```

```
struct ListNode
{
    uint packedColor;
    uint packedNormal;
    uint packedSpecular;
    uint depthAndMatterial;
    uint result;
    uint next;
};
globallycoherent RWTexture2D<uint> headBuffer;
globallycoherent RWStructuredBuffer<ListNode> frag-
mentsList;
struct NodeData
{
    uint packedColor;
    float depth;
};
float4 unpackColor(uint color)
{
    float4 output;
    output.r = float((color >> 24) & 0x000000ff) /
255.0f;
    output.g = float((color >> 16) & 0x000000ff) /
255.0f;
    output.b = float((color >> 8) & 0x000000ff) /
255.0f;
    output.a = float(color & 0x000000ff) / 255.0f;
    return saturate(output);
}
void insertionSort(uint startIndex, inout NodeData
sortedFragments[MAX_FRAGMENTS], out int counter)
{
    counter = 0;
    uint index = startIndex;
```

```
    for (int i = 0; i < MAX_FRAGMENTS; i++)
    {
        if (index != 0xffffffff)
        {
            sortedFragments[counter].packedColor =
fragmentsList[index].packedColor;
            sortedFragments[counter].depth =
f16tof32(fragmentsList[index].depthAndMatterial);
            counter++;
            index = fragmentsList[index].next;
        }
    }
    for (int k = 1; k < MAX_FRAGMENTS; k++)
    {
        int j = k;
        NodeData t = sortedFragments[k];
        while (sortedFragments[j - 1].depth < t.depth)
        {
            sortedFragments[j] = sortedFragments[j -
1];
            j--;
            if (j <= 0) { break; }
        }
        if (j != k) { sortedFragments[j] = t; }
    }
}

PS_OUTPUT main(PS_INPUT input)
{
    PS_OUTPUT output;
    uint2 upos = uint2(input.position.xy);
    uint index = headBuffer[upos];
    clip(index == 0xffffffff ? -1 : 1);
    float3 color = float3(0, 0, 0);
```

```
    float alpha = 1;
    NodeData sortedFragments[MAX_FRAGMENTS];
    [unroll]
    for (int j = 0; j < MAX_FRAGMENTS; j++)
    {
        sortedFragments[j] = (NodeData)0;
    }
    int counter;
    insertionSort(index, sortedFragments, counter);
    for (int i = 0; i < counter; i++)
    {
    float4
c=unpackColor(sortedFragments[i].packedColor);
    alpha *= (1.0 - c.a);
    color = lerp(color, c.rgb, c.a);
    }
    output.diffuse= float4(color, alpha);
    return output;
                          }
```

Приложение 10

Реализация алгоритма FXAA

Пиксельный шейдер:

```
#define EDGE_THRESHOLD_MIN 0.0312
#define EDGE_THRESHOLD_MAX 0.125
#define ITERATIONS 12
#define SUBPIXEL_QUALITY 0.75

cbuffer Null:register (b0)
{
    float2 resolution;
}
Texture2D shaderTexture;
SamplerState SampleType;

struct PixelInputType
{
    float4 position : SV_POSITION;
 float2 uv : TEXCOORD0;
};

float rgb2luma(float3 rgb)
{
    return sqrt(dot(rgb, float3(0.299, 0.587,
0.114)));
}
float3 toGamma(float3 lin)
{
    lin = max(6.10352e-5, lin);
    return min(lin * 12.92, pow(max(lin, 0.00313067),
1.0 / 2.4) * 1.055 - 0.055);
    }
```

```
float4 main(PixelInputType input) : SV_TARGET
{
    float4 diffuse;
    static const float QUALITY[12] = {
1.0,1.0,1.0,1.0,1.0,1.5, 2.0, 2.0, 2.0, 2.0, 4.0,
8.0,};
    float3 colorCenter = shaderTex-
ture.Sample(SampleType, input.uv).rgb;
    // Яркость в текущем фрагменте
    float lumaCenter = rgb2luma(colorCenter);
    // Яркость в 4 соседних пикселах
    float lumaDown =
rgb2luma(shaderTexture.Sample(SampleType, input.uv +
float2(0.0, -1.0)/ resolution).rgb);
    float lumaUp =
rgb2luma(shaderTexture.Sample(SampleType, input.uv +
float2(0.0, 1.0) / resolution).rgb);
    float lumaLeft =
rgb2luma(shaderTexture.Sample(SampleType, input.uv +
float2(-1.0, 0.0) / resolution).rgb);
    float lumaRight =
rgb2luma(shaderTexture.Sample(SampleType, input.uv +
float2(1.0, 0.0)/ resolution).rgb);
    // Поиск максимальной яркости вокруг текущего пик-
села
    float lumaMin = min(lumaCenter, min(min(lumaDown,
lumaUp), min(lumaLeft, lumaRight)));
    float lumaMax = max(lumaCenter, max(max(lumaDown,
lumaUp), max(lumaLeft, lumaRight)));
    // Разница
    float lumaRange = lumaMax - lumaMin;
    // Если яркость ниже порога или область темная, не
выполняем сглаживание
```

```
        if (lumaRange < max(EDGE_THRESHOLD_MIN,
lumaMax * EDGE_THRESHOLD_MAX)) return
float4(toGamma(colorCenter), 1);
        // считаем яркость пикселов по углам от теку-
щего.
    float lumaDownLeft =
rgb2luma(shaderTexture.Sample(SampleType, input.uv +
float2(-1.0, -1.0)/ resolution).rgb);
    float lumaUpRight =
rgb2luma(shaderTexture.Sample(SampleType, input.uv +
float2(1.0, 1.0) / resolution).rgb);
    float lumaUpLeft =
rgb2luma(shaderTexture.Sample(SampleType, input.uv +
float2(-1.0, 1.0) / resolution).rgb);
    float lumaDownRight =
rgb2luma(shaderTexture.Sample(SampleType, input.uv +
float2(1.0, -1.0) / resolution).rgb);
    // Объединим яркости ребер
    float lumaDownUp = lumaDown + lumaUp;
    float lumaLeftRight = lumaLeft + lumaRight;
    // Аналогично для углов
    float lumaLeftCorners = lumaDownLeft + lumaUpLeft;
    float lumaDownCorners = lumaDownLeft + lumaDown-
Right;
    float lumaRightCorners = lumaDownRight + lumaU-
pRight;
    float lumaUpCorners = lumaUpRight + lumaUpLeft;
    // Вычисление градиента по горизонтальной и верти-
кальной осям
    float edgeHorizontal = abs(-2.0 * lumaLeft +
lumaLeftCorners) + abs(-2.0 * lumaCenter + lumaDownUp)
* 2.0 + abs(-2.0 * lumaRight + lumaRightCorners);
```

```
    float edgeVertical = abs(-2.0 * lumaUp +
lumaUpCorners) + abs(-2.0 * lumaCenter +
lumaLeftRight) * 2.0 + abs(-2.0 * lumaDown +
lumaDownCorners);
    // определим тип края
    bool isHorizontal = (edgeHorizontal >= edgeVerti-
cal);
    // Выделение соседних текселов яркости
    float luma1 = isHorizontal ? lumaDown : lumaLeft;
    float luma2 = isHorizontal ? lumaUp : lumaRight;
    // Вычислить градиент по направлению.
    float gradient1 = luma1 - lumaCenter;
    float gradient2 = luma2 - lumaCenter;
    // Какое направление лучше?
    bool is1Steepest = abs(gradient1) >=
abs(gradient2);
    // Нормируем градиент в соответствующем направле-
нии.
    float gradientScaled = 0.25 * max(abs(gradient1),
abs(gradient2));
    // Выбор размера шага (один пиксел) в соответствии
с направлением края.
    float stepLength = isHorizontal ? 1.0/resolution.y
: 1.0/resolution.x;

    // Средняя яркость в правильном направлении.
    float lumaLocalAverage = 0.0;

    if (is1Steepest) {
        // Переключить направление
```

```
        stepLength = -stepLength;
        lumaLocalAverage = 0.5 * (luma1 + lumaCenter);
    }
    else {
        lumaLocalAverage = 0.5 * (luma2 + lumaCenter);
    }
    // Сместить UV в правильном направлении на полпик-
села.
    float2 currentUv = input.uv;
    if (isHorizontal) {
        currentUv.y += stepLength * 0.5;
    }
    else {
        currentUv.x += stepLength * 0.5;
    }
    // Вычислить смещение для каждого шага в правиль-
ном направлении.
    float2 offset = isHorizontal ?
float2(1.0/resolution.x, 0.0) : float2(0.0,
1.0/resolution.y);
    // Вычислить UVs на каждой сторне края ортогональ-
но. QUALITY позволяет действовать быстрее.
    float2 uv1 = currentUv - offset;
    float2 uv2 = currentUv + offset;
    // Считать яркость на концах сегмента и вычислить
дельту относительно местной средней яркости.
    float lumaEnd1 =
rgb2luma(shaderTexture.Sample(SampleType, uv1).rgb);
    float lumaEnd2 =
rgb2luma(shaderTexture.Sample(SampleType, uv2).rgb);
```

```
    lumaEnd1 -= lumaLocalAverage;
    lumaEnd2 -= lumaLocalAverage;
    // Если дельты больше, чем локальный градиент, то
достигли края
    bool reached1 = abs(lumaEnd1) >= gradientScaled;
    bool reached2 = abs(lumaEnd2) >= gradientScaled;
    bool reachedBoth = reached1 && reached2;
    // Если край не достигнут, двигаемся дальше.
    if (!reached1) {
        uv1 -= offset;
    }
    if (!reached2) {
        uv2 += offset;
    }
    // Если оба края не достигнуты, исследуем дальше.
    if (!reachedBoth) {

        [unroll(ITERATIONS)] for (int i = 2; i < ITER-
ATIONS; i++) {
            // При необходимости считаем яркость в 1-м
направлении и вычисляем дельту.
            if (!reached1) {
                lumaEnd1 =
rgb2luma(shaderTexture.Sample(SampleType, uv1).rgb);
                lumaEnd1 = lumaEnd1 - lumaLocalAver-
age;
            }
```

```
            // При необходимости считаем яркость
в противоположенном направлении и вычисляем дельту.
            if (!reached2) {
                lumaEnd2 =
rgb2luma(shaderTexture.Sample(SampleType, uv2).rgb);
                lumaEnd2 = lumaEnd2 - lumaLocalAver-
age;
            }
            // Если дельта яркости на краях больше те-
кущего градиента, то достигли края
            reached1 = abs(lumaEnd1) >= gradi-
entScaled;
            reached2 = abs(lumaEnd2) >= gradi-
entScaled;
            reachedBoth = reached1 && reached2;
            // Если край не достигнут, продолжаем дви-
жение, изменяя качество.

            if (!reached1) {
                uv1 -= offset * QUALITY[i];
            }
            if (!reached2) {
                uv2 += offset * QUALITY[i];
            }

            // Если оба края достигнуты, то заканчива-
ем исследование.
            if (reachedBoth) { break; }
        }
    }
```

```
    // Вычисление расстояния до краев.
    float distance1 = isHorizontal ? (input.uv.x -
uv1.x) : (input.uv.y - uv1.y);
    float distance2 = isHorizontal ? (uv2.x - in-
put.uv.x) : (uv2.y - input.uv.y);
    // В каком направлении кромка ближе?
    bool isDirection1 = distance1 < distance2;
    float distanceFinal = min(distance1, distance2);
    // Длина края.
    float edgeThickness = (distance1 + distance2);
    // UV смещение: считываем в направлении ближайшей
стороны.
    float pixelOffset = -distanceFinal / edgeThickness
+ 0.5;
    // Яркость в центре меньше, чем в среднем в окру-
жении?
    bool isLumaCenterSmaller = lumaCenter < lumaLo-
calAverage;
    // Если яркость в центре меньше, чем в окружении,
то яркость на концах должна быть положительной
    // в направлении края ближайшей стороны
    bool correctVariation = ((isDirection1 ? lumaEnd1
: lumaEnd2) < 0.0) != isLumaCenterSmaller;
    // Если изменение яркости неверное, то не смещаем.
    float finalOffset = correctVariation ? pixelOffset
: 0.0;
    // Субпиксельное смещение
    // Полновзвешенное среднее по яркости в окрестно-
стях 3x3
    float lumaAverage = (1.0 / 12.0) * (2.0 *
(lumaDownUp + lumaLeftRight) + lumaLeftCorners + lu-
maRightCorners);
```

```
    // Отношение дельты между средним общим значением
и центральной яркостью в окрестностях 3x3.
    float subPixelOffset1 = clamp(abs(lumaAverage -
lumaCenter) / lumaRange, 0.0, 1.0);
    float subPixelOffset2 = (-2.0 * subPixelOffset1 +
3.0) * subPixelOffset1 * subPixelOffset1;
    // Вычислить субпиксельное смещение, основанное
на дельте.
    float subPixelOffsetFinal = subPixelOffset2 * sub-
PixelOffset2 * SUBPIXEL_QUALITY;
    // Выбрать наибольшее из двух смещений.
    finalOffset = max(finalOffset,
subPixelOffsetFinal);
    // Вычислить итоговые UV координаты.
    float2 finalUv = input.uv;
    if (isHorizontal) {
        finalUv.y += finalOffset * stepLength;
    }
    else {
        finalUv.x += finalOffset * stepLength;
    }
    // Считать цвет по новым UV координатам и исполь-
зовать его.
    float3 finalColor = shaderTex-
ture.Sample(SampleType, finalUv).rgb;
    return float4(toGamma(finalColor), 1.0);
}
```

Учебное электронное издание

САМОЙЛОВ Сергей Александрович

ПРОГРАММИРОВАНИЕ ТРЕХМЕРНОЙ ГРАФИКИ СРЕДСТВАМИ БИБЛИОТЕКИ DIRECTX 11

Учебное пособие

Редактор Е. А. Платонова

Технический редактор Ш. Ш. Амирсейидов

Компьютерная верстка Л. В. Макаровой

Корректор О. В. Балашова

Выпускающий редактор А. А. Амирсейидова

Системные требования: Intel от 1,3 ГГц; Windows XP/7/8/10; Adobe Reader; дисковод CD-ROM.

Тираж 9 экз.

Издательство Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых.
600000, Владимир, ул. Горького, 87.