МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ» (НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Кафедра информационно-измерительных систем

Артиков Артур Неъматжанович

Разработка системы автоматизированной генерации шейдеров для реалистичной визуализации трехмерных сцен

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ
по направлению высшего профессионального образования

230100.68 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Тема диссертации утверждена распоряжением по НГУ № 8 от «11» января 2012 г.
Тема диссертации скорректирована распоряжением по НГУ № 535 от «14» декабря 2012 г.

Руководитель

Долговесов Б. С.

к.т.н., зав. лаб. ИАиЭ СО РАН

Новосибирск, 2013 г.

Содержание

Введение

В настоящее время широкое распространение получили системы, предназначенные для реалистичной визуализации интерактивных трехмерных сцен. Такие системы являются важным элементом многих графических приложений, таких как компьютерные тренажеры, виртуальные видеостудии, различные обучающие демонстрации. В Лаборатории синтезирующих систем визуализации ИАиЭ СОРАН разрабатывается модуль визуализации, основной областью применения которого являются системы виртуальной реальности (СВР). В таких системах видеоизображения реальных актеров комбинируются с синтезированным изображением виртуальной сцены. Важную роль в СВР играет реалистичность визуализации.

Совокупность визуальных свойств поверхности трехмерного объекта называют *материалом* объекта. К этим свойствам относятся цвет, прозрачность, уровень блеска, способность отражать или преломлять свет, рельефность, самосвечение и др. Некоторые свойства материала, такие как цвет или рельефность, задаются двухмерными изображениями – текстурами, а другие – числовыми значениями. Часть свойств материала может отсутствовать (к примеру, не все материалы преломляют свет, и не у всех есть текстура рельефности). Помимо материала на внешний вид объекта влияют свойства всей сцены в целом. К ним относятся параметры источников света, наличие тумана и др.

Для визуализации трехмерных объектов используются шейдеры – программы, исполняемые на процессоре видеокарты. На основе описания геометрии объекта, свойств материала и сцены видеокарта с помощью заданного шейдера вычисляет изображение объекта. Важной задачей при разработке системы визуализации является написание самих шейдеров. Один из возможных подходов к решению данной задачи – написать один универсальный шейдер и использовать его для всех объектов сцены. Однако было выяснено, что для разрабатываемой системы визуализации этот подход не приемлем. Количество свойств материалов и сцены в данной системе велико (несколько десятков). Некоторые свойства материала могут отсутствовать. Количество источников света на сцене и их тип также варьируются. Для того чтобы учесть все эти случаи, в коде шейдера пришлось бы использовать большое количество операций условного перехода, что привело бы к значительному снижению производительности. Кроме того, существует ограничение на количество инструкций, допустимое в одном шейдере. Исходя из вышесказанного, было решено использовать несколько разных шейдеров, и в зависимости от ситуации на сцене и материала рисуемого объекта выбирать нужный шейдер. Однако написание всех необходимых шейдеров вручную не представляется возможным, поскольку количество возможных комбинаций очень велико. В связи с чем, необходимо

разработать систему, позволяющую по заданным свойствам материала и сцены генерировать код шейдера.

Для достижения этой цели были поставлены следующие задачи:

1) Согласование и уточнение требований, предъявляемых к разрабатываемой системе.
2) Исследование существующих подходов, применяемых в современных системах визуализации.
3) Разработка формата описания материалов.
4) Разработка метода автоматизированной генерации шейдеров.
5) Реализация системы генерации шейдеров.
6) Разработка интерфейса для взаимодействия системы генерации шейдеров с системой визуализации.
7) Написание примеров, демонстрирующих работоспособность системы

Использование автоматизированной генерации позволяет свести к минимуму количество лишних инструкций в коде шейдеров и существенно повысить скорость визуализации объектов.

Хотя способов автоматизированной генерации программного кода существует уже достаточно много, генерация кода шейдеров имеет свою специфику и требует пересмотра существующих подходов. Автором работы изучены подходы, применяемые в системах генерации шейдеров, и разработан метод генерации шейдеров, в котором пиксельный шейдер строится на основе шейдера поверхности и шейдера модели освещения, а также применяется условная компиляция кода шейдеров. Особенность данного подхода заключается в том, что он позволяет генерировать высокопроизводительные шейдеры и в то же время обеспечивает расширяемость системы.

Работа состоит из трех глав. В первой главе описывается предметная область, требования к системе и существующие методы генерации шейдеров. Во второй главе рассказывается о формате описания материалов и разработке метода генерации шейдеров. Третья глава посвящена программной реализации системы генерации шейдеров и примерам использования этой системы.

ГЛАВА 1. Анализ предметной области

1.1 Система реалистичной визуализации

В Лаборатории синтезирующих систем визуализации ИАиЭ СОРАН разрабатывается система реалистичной визуализации трехмерных сцен. Эта система является системой общего назначения, и будет применяться в различных интерактивных приложениях: симуляторах, тренажерах, компьютерных играх, различных обучающих демонстрациях, приложениях для научной визуализации. Однако ее основное назначение – это системы виртуальной реальности (СВР) [1].

В СВР визуализация осуществляется следующим образом. Реального актера снимают на видеокамеру на одноцветном фоне. Полученное видеоизображение обрабатывается алгоритмом хромакеинга, который удаляет одноцветный фон, а затем помещают в виртуальную трехмерную сцену. При визуализации сцены виртуальную камеру располагают так, чтоб ее положение соответствовало реальной камере, из которой был снят актер, за счет чего достигается эффект полного совмещения актера и виртуальной среды. Кроме этого, оператор или же сам актер может взаимодействовать с объектами сцены с помощью различных устройств управления (мышь, клавиатура, система Kinect, джойстик, тачскрин и др.): перемещать камеры, источники освещения и трехмерные модели, менять их свойства. Необходимость интерактивного взаимодействия накладывает требования к производительности системы – около 60 кадров в секунду, а производительность всей системы в большой степени определяется производительностью системы визуализации. Высокая скорость визуализации достигается за счет использования графических ускорителей.

Кроме производительности, большое значение в СВР играет реалистичность визуализации. Изображения виртуальной сцены, полученные в результате визуализации, должны быть максимально приближены по качеству к фотографиям реальных объектов. Для придания сцене реалистичности объектам сцены назначаются *материалы* [2]. Материалом трехмерного объекта будем называть совокупность визуальных свойств его поверхности. Эти свойства включают в себя цвет поверхности, рельефность, прозрачность, самосвечение, способность объекта отражать или преломлять свет. Так же к ним относится модель освещения, которая характеризует, как поверхность объекта взаимодействует с лучами света. Особенностью разрабатываемой системы визуализации в отличие от систем, предназначенных для компьютерных играх, является наличие большого количества свойств материалов (несколько десятков) и широкие возможности по комбинированию этих свойств. К примеру, свойства материала задаются как числовыми значениями, так и текстурами, а в одном материале может быть использовано

до 8 текстур одновременно. Для сравнения в системе разработки игр Unity разработчику даются шаблоны материалов с сильно ограниченными возможностями настройки и 2-3 текстурами в каждом. Помимо свойств материала на внешний вид объекта влияют свойства всей сцены: источники света (точечные, направленные, конусные), туман, фоновая подсветка.

Для визуализации объектов сцены используют шейдеры, небольшие программы, исполняемые на графическом процессоре. Существуют высокоуровневые языки для написания шейдеров: HLSL (для DirectX), GLSL (для OpenGL), Cg [3]. Автором работы используется язык Cg, программы на котором могут быть транслированы в код HLSL или GLSL с помощью программной библиотеки NVidia Cg Toolkit. Шейдер реализует одну из стадий конвейера визуализации. Наиболее часто используются вершинные и пиксельные шейдеры. В вершинном шейдере происходит преобразование вершин объекта из системы координат объекта в пространство экрана, а также обработка некоторых других атрибутов вершин, таких как цвет, нормаль, текстурные координаты и др. Пиксельный шейдер вызывается для каждого пикселя, получаемого при растеризации трехмерного объекта. На вход пиксельного шейдера приходят линейно интерполированные значения атрибутов, содержащихся в вершинах растеризуемого треугольника, а на выход выдается цвет обрабатываемого пикселя.

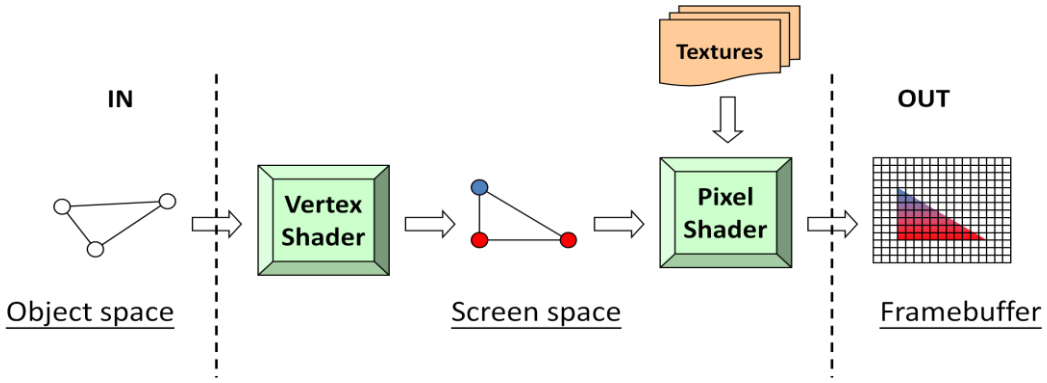

Рисунок 1: Вершинный и пиксельный шейдеры

Так же вершинные/пиксельные шейдеры могут принимать дополнительные параметры, общие для всех вершин/пикселей объекта. Примерами таких параметров могут служить мировая матрица объекта, позиция источника света, степень прозрачности, цвет или текстура поверхности. Большинство параметров связано с тем или иным свойством материала или сцены. Система визуализации устанавливает параметры вершинного и пиксельного шейдеров перед визуализацией объекта, тем самым влияет на внешний вид этого объекта. Язык Cg позволяет объединить в одном файле вершинный и пиксельный шейдеры, а так же их параметры. В дальнейшем такое объединение будем для удобства называть просто шейдером.

1.2 Необходимость генерации шейдеров

Важной задачей при разработке системы визуализации является написание шейдеров, поскольку возможности системы во многом определяются набором функций, реализуемых с помощью шейдеров. Производительность визуализации также очень сильно зависит от того, насколько эффективно реализованы шейдеры (особенно пиксельные шейдеры, поскольку в них сосредоточена большая часть чтений из видеопамяти и вычислений).

Принципы написания программ для графического процессора очень сильно отличаются от тех, которые используются при разработке программ, исполняемых на центральном процессоре. Программы для центрального процессора, как правило, универсальны, то есть их программный код реализует все необходимые варианты поведения программы и подходит для всех допустимых входных значений. Написание универсального шейдера, который бы реализовывал все материалы, использующиеся в системе, и при этом исполнялся быстро и на всем целевом аппаратном обеспечении, как правило, является неразрешимой задачей. На это есть несколько причин.

Во-первых, существует ограничение на количество инструкций, допустимых в шейдере (к примеру, пиксельные шейдеры модели 2.0 поддерживают до 64 арифметических инструкций и до 32 инструкций чтения из текстур). На современных видеокартах максимальное допустимое количество инструкций существенно больше, но для встроенных видеокарт и мобильных устройств эта проблема все еще актуальна.

Во-вторых, универсальный шейдер будет содержать большое количество операторов условного перехода и циклов, наличие которых существенно снижает производительность вычислений на графическом процессоре. Производительность теряется как на самих операторах условного перехода, так и на установке флагов, управляющих этими условиями. Параметры шейдера, которые используются внутри блока условного оператора, будут устанавливаться в шейдер в любом случае, даже если само условие не выполняется, поскольку программный интерфейс для работы с шейдерами не предоставляет возможности проверить, потребуется ли параметр при вычислениях. Для передачи интерполированных значений атрибутов из вершинного шейдера в пиксельный используется структура с некоторым набором полей. В случае универсального шейдера в эту структуру придется добавить все поля, которые могут понадобиться в пиксельном шейдере, а это: позиция в экранном и мировом пространстве, текстурная координата, нормаль, тангент и бинормаль (два последних атрибута используются, если включена визуализация микрорельефа). Для простейшего материала, который окрашивает объект один цвет без учета освещения, в структуре достаточно лишь атрибута «позиция в

экранном пространстве», однако попытка написать универсальный шейдер приведет к тому, что все остальные вышеперечисленные атрибуты также будут передаваться из вершинного шейдере в пиксельный, то есть возникнут накладные расходы на их интерполяцию.

Из вышесказанного следует, что для реализации высокопроизводительной системы визуализации необходимо написать несколько шейдеров и при визуализации объекта в зависимости от его материала и свойств сцены выбирать наиболее подходящий шейдер. Типичные случаи, приводящие к появлению новых шейдеров это:

- Наличие или отсутствие текстуры
- Разное количество источников света

Несложно посчитать, что если материал поддерживает до восьми текстур, каждая из которых может быть отключена, и до восьми источников света, то общее количество возможных комбинаций (а значит и шейдеров) составит $2^8 * 9 = 2304$. Эта цифра будет еще больше, если добавить поддержку различных типов источников света, флаг наличия тумана и т.д. Написать такое большое количество шейдеров вручную невозможно, поэтому в работе предлагается использовать автоматизированную генерацию шейдеров.

Дополнительное преимущество, которое дает автоматизированная генерация шейдеров, это расширяемость системы. В случае с универсальным шейдером каждое добавление новых возможностей требует внесения исправлений в этот шейдер. Если шейдеров несколько, но они пишутся вручную, то в них неизбежно будут дублирующиеся участки кода, соответственно исправления нужно будет производить в нескольких местах. Использование генерации шейдеров позволяет избежать этих проблем и дает возможность добавлять новые способы расчета характеристик материала без внесения изменений в саму систему.

1.3 Требования к системе генерации шейдеров

Автором работы были поставлены следующие требования к системе генерации шейдеров:

1. *Шейдер должен генерироваться на основе свойств материала и сцены.* Стоит отметить, что поскольку свойства сцены так же влияют на генерацию шейдеров, то на один и тот же материал могут приходиться несколько шейдеров (к примеру, когда объекты с одинаковым материалом освещаются разным количеством источников света). Более того, генерацию в таком случае нельзя перенести на этап загрузки материала, поскольку в этот момент объекты не помещены на сцену и неизвестно, какие источники воздействуют на них.
2. *Генерируемый шейдер должен быть эффективным.* Другими словами, в коде получаемых шейдеров не должно быть лишних вычислений и операций считываний из текстур, а границы циклов должны быть константами (такие циклы компилятор шейдеров может лучше оптимизировать путем разворачивания циклов).
3. *Для любого целочисленного и булевого свойства материала или сцены должна быть возможность указать, нужно ли для различных значений этого свойства генерировать уникальный шейдер.* Это требование связано с тем, что значения некоторых из свойств можно изменять в пределах одного шейдера без какого-либо снижения производительности (либо с ничтожно малым снижением).
4. *Генерируемый шейдер должен поддерживать следующие текстуры: diffuse, ambient, specular, reflect, lightmap, opacity, bump, emission.* Названия этих текстур и их влияние на внешний вид объекта взяты по аналогии с широко используемым редактором трехмерных моделей 3ds max, экспорт из которого поддерживается разрабатываемой системой визуализации. Результат визуализации с помощью сгенерированого шейдера должен быть идентичен изображению, полученному визуализацией в 3ds max.
5. *Генерация шейдеров по требованию.* Система генерации должна по запрашиваемым свойствам материала и сцены генерировать шейдер. Вариант, когда система генерирует все шейдеры для всех возможных вариантов свойств, недопустим, поскольку количество различных комбинаций велико (несколько тысяч).
6. *Расширяемость* - должна быть возможность добавлять новые способы расчета характеристик поверхности объекта и новые модели освещения. Добавление этих возможностей не должно приводить к дублированию кода. Расчет теней и тумана должен поддерживаться всеми материалами.

1.4 Существующие методы генерации шейдеров

Автором были исследованы подходы, применяемые в современных системах визуализации для генерации шейдеров. Существующие подходы можно разделить на две группы: использование убершейдеров и сборка шейдеров из фрагментов.

Убершейдером называют шейдер, который реализует сразу несколько (зачастую все) возможности материалов, поддерживаемых системой визуализации. В широком смысле этого слова убершейдером можно назвать даже шейдер, в котором для выбора необходимого способа расчета цвета используются операторы условного перехода, однако, как было сказано ранее, такой подход приводит к снижению производительности, а потому может быть применен только в простых системах визуализации, где количество свойств системы невелико. Гораздо чаще под применением убершейдеров подразумевают условную компиляцию шейдеров. Такие убершейдеры содержат в своем коде директивы препроцессора. Компилируя один и тот же убершейдер с разными флагами пропроцессора, можно получить разные шейдеры. Данный подход описан в докладе конференции КРИ «Промышленные убершейдеры» представителями компании Saber Interactive [4], специализирующейся на разработке компьютерных игр. В их системе визуализации один и тот же убершейдер используется практически для всех отображаемых объектов (исключение составляют частицы и элементы пользовательского интерфейса). Использование генерации на основе убершейдеров приводит к появлению большого количества шейдеров, равного количеству сочетаний флагов, подаваемых препроцессору. Компиляция шейдеров достаточно долгая операция, поэтому важной задачей при использовании данного подхода является организация кеширования шейдеров. Преимуществами данного подхода являются простота реализации (компилятор шейдеров уже содержит в себе препроцессор) и эффективность получаемых шейдеров (поскольку использование препроцессора позволяет исключить из шейдера лишние инструкции).

Сборка из фрагментов – это большой класс способов генерации шейдеров. Идея данного подхода заключается в том, что задается набор шейдерных фрагментов (функций на языке шейдеров), а в генераторе шейдеров определяются правила, по которым из этих фрагментов собирается код шейдера. В системах визуализации, изученных автором, представлены различные способы задания фрагментов и правила их объединения.

В системе UDK [5], предназначенной для разработки интерактивных приложений, фрагменты – это достаточно низкоуровневые функции, такие как считывание значений из текстур и различные операции смешивания цветов, организованные в библиотеку фрагментов. Для UDK разработана графическая среда редактирования материалов, где

разработчик может выбирать фрагменты, которые будут использованы в шейдере материала, объединять входы и выходы фрагментов дугами. Фактически описание материала в этой системе – это программа на графическом высокоуровневом языке. Данный подход очень гибок и позволяет описывать сложные материалы, но в некоторых случаях удобнее задавать материал набором свойств, а не в виде графа. Так же этот способ не подходит для генерации шейдера расчета освещения от произвольного количества источников света, поскольку с помощью графа фрагментов нельзя задать цикл по источникам (в связи с этим расчет освещения в UDK задается отдельно от графа материала).

В Unity [6], системе для создания интерактивных приложений, фрагменты принимают на вход структуры некоторого фиксированного формата и могут быть одного из следующих типов: vertex (задает преобразование формы объекта), surface (задает способ расчета цвета и нормали поверхности), lighting (задает способ расчета освещения). В описании материала можно указать по одному фрагменту каждого типа из существующих. Есть возможность написать собственные фрагменты на языке Cg.

В программе для монтажа видео Adobe Premiere, поддерживающей визуализацию трехмерных сцен, материалы объекта задаются с помощью слоев. Каждый слой описывает функцию расчета цвета поверхности, при визуализации эти слои смешиваются с помощью заданных операций или по маске.

Преимуществом методов, основанных на сборе шейдеров из фрагментов, является гибкость и расширяемость системы. Большинство таких систем позволяют описывать собственные фрагменты (в одних системах с произвольными входами и выходами, в других с жестко заданными) и предоставляют широкие возможности для комбинирования фрагментов. Недостатком подхода является то, что генерируемый шейдер не всегда получается эффективным, поскольку по заданному набору фрагментов генерируется лишь один шейдер, а это значит, что он не оптимизирован под конкретное количество источников света, и в нем не учитывается, что влияние какой-либо из текстур может быть отключено.

Автором работы предлагается комбинированный метод генерации шейдеров, который позволяет получать эффективные шейдеры и в то же время обеспечивает расширяемость системы.

ГЛАВА 2. Разработка метода генерации шейдеров

2.1 Формат описания материалов

Материалы в системе визуализации описываются с помощью скриптового языка XQL (eXtended Query Language), разработанного в Лаборатории синтезирующих систем визуализации ИАиЭ СОРАН. Этот язык служит как для описания данных, так и для их изменения в процессе работы программы. Данные представляются в виде дерева и хранятся в XQL-базе, хранилище доступном всем модулям системы [7]. Узлы дерева имеют имена и могут содержать данные примитивных типов (целые и вещественные числа, строки). У узла допускается произвольное количество дочерних узлов. Адрес узла – это строка, содержащая перечисление имен узлов в пути от корня дерева до этого узла, разделенных точкой. Адрес однозначно определяет узел и может быть использован для задания ссылки на узел. Язык XQL позволяет описывать объекты. Предполагается, что некоторые узлы обозначают объекты, а их дочерние узлы – свойства этих объектов.

Описания данных на языке XQL состоят из адреса узла, знака присваивания и значения. Например:

```
MATERIAL.wood.AMBIENT = #FF42C1
MATERIAL.wood.MAP.DIFFUSE = IMAGE.wood
```

Это же описание можно записать короче, используя фигурные скобки:

```
MATERIAL.wood = {
      AMBIENT = #FF42C1
      MAP.DIFFUSE = IMAGE.wood
}
```

У материалов, разрабатываемой системы, существуют следующие свойства:

`AMBIENT, DIFFUSE, SPECULAR` – фоновый, диффузный и зеркальный цвет

`SELF` – коэффициент самосвечения

`SHIN_STR, POWER` – коэффициенты, управляющие силой и размером блика

`TRANSP` – прозрачность

Кроме этого можно задать текстурные карты (указываются в дочерних узлах узла `MAP`):

`AMBIENT, DIFFUSE, SPECULAR` – карты фонового, диффузного и зеркального цвета

`OPACITY` – маска прозрачности

`LIGHTMAP` – карта предрассчитанного статического освещения

`BUMP` – карта нормалей, используется для расчета микрорельефа

`EMISSION` – карта, задающая самосвечение

`REFLECT` – карта отражений

У каждой карты могут быть заданы ее вес (`WEIGHT`) и преобразования текстурных координат: `SCL` – масштаб, `POS` – сдвиг, `ROT` – поворот. Помимо перечисленных свойств, в материале могут быть указаны другие свойства, с произвольными именами. Эти свойства используются для нестандартных материалов с нестандартными шейдерами.

2.2 Условная компиляция шейдеров

Как уже было упомянуто ранее, количество арифметических инструкций и операций считывания из текстур допустимое в шейдере ограничено. Так же отсутствие последовательного выполнения кода шейдера, то есть наличие операторов перехода по условию, задаваемому извне с помощью параметра шейдера, или циклов по диапазону значений, известному только на этапе исполнения, приводит к снижению производительности визуализации. Для решения этих проблем автором предлагается использовать условную компиляцию, то есть генерацию шейдеров на основе убершейдера, содержащего директивы препроцессора.

Наиболее часто используются директива #ifdef. Ниже приведен пример директивы препроцессора, в котором в зависимости от наличия у материала диффузной текстуры, осуществляется либо считывание из этой текстуры, либо берется цвет, заданный вектором вещественных чисел.

```
#ifdef  HAS_DIFFUSE_MAP
    diffuse = tex2D(diffuseMap, tc);
#else
    diffuse = diffuseColor;
#endif
```

Данный подход можно расширить, добавив помимо флагов препроцессора, принимающих два значения (определен / неопределен) целочисленные константы, определяемые на этапе компиляции, что можно использовать, например, для задания количества источников света.

Значения флагов и целочисленных констант, управляющих генерацией шейдеров, приходят из модуля визуализации и определяются на основе свойств используемого материала и сцены. Перед генерацией эти значения устанавливаются в убершейдер, для этого система генерации добавляет к коду убершейдера команды определения флагов и констант. Например:

```
#define HAS_DIFFUSE_MAP
#define LIGHT_COUNT 4
```

После этого вызывается компиляция убершейдера. В случае если используется Cg, сначала идет трансляция кода в GLSL или HLSL и роль препроцессора выполняет транслятор, а уже затем происходит компиляция шейдеров.

Нетрудно заметить, что количество шейдеров, которые могут быть сгенерированы равно: $2^B * C_1 * \ldots * C_N$, где B это количество флагов, N – количество целочисленных констант, а C_i количество значений, которые может принимать i-я константа.

Если убершейдер содержит вложенные команды препроцессора, то возможна ситуация, когда для разных флагов будет сгенерирован одинаковый код шейдера. Например в случае такой организации кода:

```
#ifdef  A
    …
    #ifdef  B
    …
    #endif
    …
#endif
```

если флаг A не определен, то не зависимо от того, какое значение принимает B, весь код между «#ifdef A» и последним «#endif» будет исключен препроцессором. Такие ситуации надо обрабатывать для того, чтобы уменьшить количество используемых шейдеров и уменьшить время, затрачиваемое на компиляцию и линковку шейдеров. Один из способов сделать это – посимвольно сравнивать получаемый шейдер с уже сгенерированными раньше и в случае совпадения исключать его, а для ускорения сравнения сначала сравнивать шейдеры по хэшам.

Существуют различные способы организовать генерацию шейдеров, основанную на условной компиляции. Автором предлагается генерировать шейдеры по требованию (в противовес способу, когда заранее генерируются все возможные шейдеры). Шейдер генерируется тогда, когда он впервые потребуется для визуализации какого-либо объекта. Для того чтобы большая часть шейдеров генерировалась в момент запуска программы, первый кадр визуализации осуществляется без отсечения объектов сцены по пирамиде видимости. Это позволяет уменьшить количество вызовов генерации шейдеров во время взаимодействия пользователя с программой, что необходимо в силу того, что компиляция большого количества шейдеров может занимать достаточно много времени. Нечастая компиляция небольшого количества шейдеров во время взаимодействия с пользователем допустима. Такие случаи возникают, например, когда у материала в процессе работы программы включают или отключают влияние текстуры, либо когда меняется количество источников света, освещающих объект.

Преимущество генерации шейдеров по требованию заключается в том, что она не приводит к появлению лишних, неиспользуемых шейдеров. Однако даже в этом случае количество шейдеров, появляющихся в системе, может быть велико. Большое количество шейдеров приводит к большому времени загрузки программы, кроме этого частая смена шейдеров во время визуализации повышает нагрузку на центральный процессор. С этим можно бороться путем правильной организации кода убершейдера. Далеко не все

целочисленные и булевы значения стоит делать константами и флагами времени компиляции. Некоторые из них могут задаваться через параметры шейдеров, и это не приведет к снижению производительности. К примеру, экспериментальным путем было выяснено, что свойство материала ALPHA, которое определяет, нужно ли учитывать прозрачность диффузной текстуры, не имеет смысла делать флагом препроцессора, так как это приводит к увеличению возможного количества шейдеров вдвое без какого-либо выигрыша производительности. Аналогично было решено всегда рассчитывать преобразования текстурных координат (даже если будет происходить умножение на единичные матрицы), а не делать флаги наличия текстурных преобразований, поскольку накладные расходы были незначительны. Подобные решения позволили найти баланс между количеством генерируемых шейдеров и производительностью этих шейдеров.

Для проверки эффективности генерации с помощью условной компиляции автором было написано две версии шейдера, реализующего все необходимые возможности стандартного материала:

1. Шейдер, в котором флаги наличия текстур и количество источников света передаются через параметры шейдера (универсальный шейдер)
2. Убершейдер, в котором неиспользуемые возможности материала могут быть отключены с помощью препроцессора.

Схематично эти шейдеры выглядят так:

1. универсальный шейдер

```
bool hasDiffuseMap;
bool hasBumpMap;
…
bool hasOpacityMap;
int lightCount;

float4 main(INPUT in)
{
  if(hasDiffuseMap) {
  … //считывание цвета из текстуры
  }

  if(hasBumpMap) {
  … // расчет микрорельефа
  }

  int i;
  for(i = 0; i < lightCount; i++){
    … // расчет освещения
  }

  … //аналогично остальные текстуры

}
```

2. Убершейдер

```
float4 main(INPUT in)
{
  #ifdef HAS_DIFFUSE_MAP
  … //считывание цвета из текстуры
  #endif

  #ifdef HAS_BUMP_MAP
  … // расчет микрорельефа
  #endif

  int i;
  for(i = 0; i < LIGHT_COUNT; i++){
    … // расчет освещения
  }

  … //аналогично остальные текстуры

}
```

Была исследована производительность обоих шейдеров на простейшем материале, у которого нет ни одной текстуры и выключено освещение (именно в этом случае накладные расходы от ненужных инструкций в шейдере проявляются наиболее сильно). На тестовой сцене, состоящей из 10 объектов с одинаковым материалом, было замерено время визуализации и были получены следующие результаты:

1. *Универсальный шейдер: 25 мс*
2. *Убершейдер: 15.4 мс*

Характеристики компьютера: процессор Intel Core 2 Duo, видеокарта Geforce 9300 GS, разрешение экрана 1280x720.

Как можно видеть, универсальный шейдер проигрывает по скорости убершейдеру более чем в 1.5 раза, и производительность его работы на данной сцене не позволяет уложиться в допустимые 16.6 мс (60 кадров в секунду), что неприемлемо для интерактивных приложений. Аналогичный прирост производительности (в 1.5 – 2 раза в зависимости от сложности сцены) был получен за счет использования условной компиляции и на реальных сценах (рис. 2).

Рисунок 2: Одна из сцен, на которых исследовалась производительность шейдеров

Код, описанного выше убершейдера, содержит около 400 строк и 30 параметров, что достаточно много для шейдера. В связи с таким размером его дальнейшая доработка и добавление новых возможностей привела бы к еще более сильному усложнению кода и проблемам с сопровождением этого кода. Для решения этой проблемы было принято решение разработать еще один механизм генерации шейдеров, а именно генерацию на основе высокоуровневых шейдеров.

2.3 Генерация на основе высокоуровневых шейдеров

Чтобы система визуализации была расширяемой, необходимо реализовать в системе возможность описывать различные способы вычисления характеристик поверхности и различные модели освещения. К характеристикам поверхности относятся диффузный цвет, зеркальный цвет, нормаль и самосвечение поверхности. В стандартном материале эти характеристики считываются из текстур, используя текстурные координаты, которые хранятся в вершинах. В некоторых случаях требуются более сложные варианты расчета характеристик поверхности, использующие дополнительные текстуры или процедурное текстурирование. Модель освещения описывает, как поверхность объекта взаимодействует с лучами света. Наиболее часто используется модель освещения по Фонгу, но для более качественного результата визуализации бывает целесообразно использовать и другие модели освещения (см. 3.2.3 модель освещения Кука-Торренса).

Идею генерации с помощью высокоуровневых шейдеров проще показать на следующем примере. Представим, что нужно написать на некотором объектно-ориентированном языке (например, C++) класс материала, который будет поддерживать несколько способов расчета освещения, условно назовем эти способы LIGHTING_A, LIGHTING_B, LIGHTING_C. Один из вариантов сделать это – использовать оператор выбора (switch):

```
class Material
{
      void lighting()
      {
            switch(lightingType)
            {
                  case LIGHTING_A:
                  … // вычисляем освещение способом A
                  break;
                  case LIGHTING_B:
                  … // вычисляем освещение способом B
                  break;
                  case LIGHTING_C:
                   … // вычисляем освещение способом C
                  break;
            }
      }

      void calculate()
      {
            … // остальные вычисления
            lighting();
      }

      LightingType lightingType;
};
```

Минус такого подхода очевиден, при добавлении новых типов освещения необходимо вносить исправления в функцию `lighting`, добавляя в оператор выбора новые варианты. ООП-подход диктует нам другое, более удобное и легко расширяемое решение. Создадим базовый класс материала с виртуальным методом `lighting`, а различные способы расчета освещения реализуем в его наследниках.

```
class Material
{
      virtual void lighting() = 0;
      void calculate()
      {
            … // остальные вычисления

            lighting();
      }
};
class Material_LightingA : public Material
{
      void lighting()
      {
      … // вычисляем освещение способом A
      }
};

… // аналогично для способов B и C
```

В ООП такой прием называется шаблонным методом (template method) [8]. Аналогичным образом, если потребуется реализовать еще и несколько способов расчета характеристик поверхности, то в класс материала будет добавлен метод surface с последующей реализацией его в наследниках. Посредством множественного наследования можно сочетать классы с различными методами `surface` и `lighting`, тем самым получать разные материалы:

```
class Material_LighingA_SurfaceE : public  Material_LightingA,
                                           Material_SurfaceE
```

Точно такой же подход можно использовать и для реализации материалов с помощью шейдеров. Разница заключается в том, что в языках для написания шейдеров нет классов и наследования. Вместо этого можно применить сборку кода шейдера из фрагментов. Введем две функции на языке Cg со следующими сигнатурами:

```
SurfaceOutput surface(SurfaceInput input)
LightingOutput lighting(LightingInput input)
```

где входные и выходные структуры имеют вид:

```
struct SurfaceInput
{
      float2 texCoord;
      float2 diffuseTexCoord;
      float3 worldPosition;
      float3 objectPosition;
      float3 normal;
      float3 tangent;
      float3 binormal;
};
```

```
struct SurfaceOutput
{
      float4 diffuse;
      float4 specular;
      float3 normal;
      float4 emission;
};
```

```
struct LightingInput
{
      float3 normal;
      float3 lightDir;
      float3 viewDir;
};
```

```
struct LightingOutput
{
      float diffuseK;
      float specularK;
};
```

Функция `surface` принимает на вход данные, полученные после интерполяции вершинных атрибутов, и возвращает рассчитанные характеристики поверхности. Функция `lighting` служит для задания модели освещения. Ей на вход подается нормаль поверхности, направление взгляда и направление луча света, а выход возвращается сила диффузной и зеркальной составляющих освещения. Эти функции по аналогии с вершинным и пиксельным шейдерами выполняют определенную часть работы по формированию изображения сцены, но манипулируют более высокоуровневыми понятиями – не вершинами и пикселями, а характеристиками поверхности и лучами света. Поэтому будем называть их высокоуровневыми шейдерами. Функция `surface` – это шейдер поверхности, а `lighting` – шейдер модели освещения. Помимо входных структур высокоуровневый шейдер может использовать в своих вычислениях дополнительные параметры, глобальные по отношению к нему. Имена параметров шейдера поверхности начинаются с префикса `s_`, а шейдера модели освещения с `l_`. Это сделано для того, чтобы избежать возможного конфликта имен при генерации шейдеров. Из высокоуровневых шейдеров генерируется шейдер с помощью шаблона. Этот шаблон схематично выглядит следующим образом:

```
<объявление параметров шейдера>
<объявление структур SurfaceOutput, SurfaceInput,
                     LightingOutput, LightingInput>
[@surface]
[@lighting]
```

```
VertexToPixel vertexShader(VertexShaderInput input)
{
      VertexToPixel output;
      < заполнение output>
      return output;
}

float4 pixelShader (VertexToPixel input)
{
      float4 result;
      SurfaceInput surfaceInput;
      <заполнение surfaceInput>
      SurfaceOutput surfaceOutput = surface(surfaceInput);

      for(int i = 0; i < LIGHT_COUNT; i++)
      {
      LightingInput lightingInput;
      <заполнение lightingInput>
      LightingOutput lightingOutput = lighting(lightingInput);
            <расчет теней>
            <добавление к result вклада от источника>
      }

      <обработка дополнительных текстур: отражений, прозрачности>

      return result;
}
```

В этом шаблоне есть вызовы функций `surface` и `lighting`, но нет их реализаций. Вызов функции `surface` осуществляется один раз в начале пиксельного шейдера, функция `lighting` вызывается для каждого источника света. Строками `[@surface]` и `[@lighting]` указаны места в шаблоне, куда нужно подставить реализации функций `surface` и `lighting` при генерации шейдера. Сначала из описания материала считывается, какой шейдер поверхности и шейдер модели освещения использовать, затем генератор шейдеров подставляет код этих шейдеров в шаблон и возвращает полученный шейдер.

Шейдер, полученный посредством генерации на основе высокоуровневых шейдеров, является убершейдером, поскольку код шаблона содержит в себе директивы препроцессора, и допускается использование этих директив в коде высокоуровневых шейдеров. Таким образом, генерация шейдеров происходит в два этапа. Сначала из шейдера поверхности и шейдера модели освещения строится убершейдер, затем путем условной компиляции получается шейдер, который и будет использован при визуализации объекта (рис. 3).

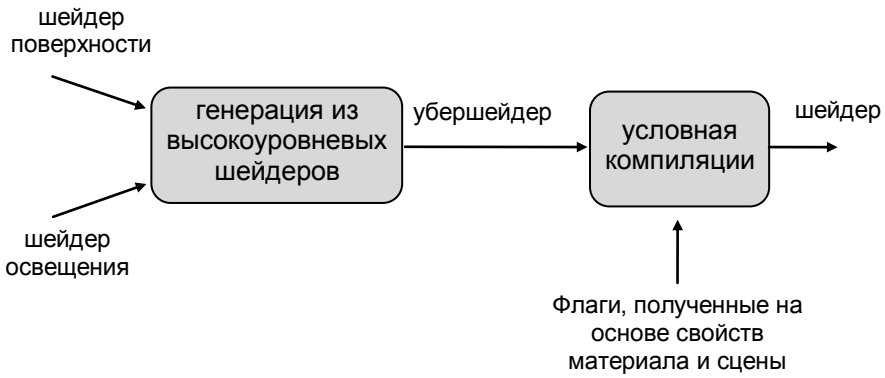

Рисунок 3: Генерация шейдера

Любой шейдер поверхности может быть скомбинирован с любым из шейдеров модели освещения, что дает широкие возможности для настройки внешнего вида объектов. Для добавления новых возможностей в систему пишутся только высокоуровневые шейдеры. Весь остальной шейдерный код генерируется автоматически.

ГЛАВА 3. Программная реализация

3.1 Система автоматизированной генерации шейдеров

Система автоматизированной генерации шейдеров состоит из двух модулей: модуль высокоуровневых шейдеров (Highlevel shader) и модуль условной компиляции (Ubershader). Модули написаны на языке C++ и предназначены для генерации шейдеров на языке Cg. Взаимодействие модулей с системой визуализации (RenderOpenGL) осуществляется через XQL-базу. Для этого используется концепция XQL-плагинов. На любой узел базы можно установить плагин (обработчик событий). События бывают следующих типов: создание узла, запись в узел, чтение узла, удаление узла. Обработчики вызываются как при возникновении события в узле, на который он был установлен, так и на дочерних узлах любого уровня глубины. Этот механизм позволяет осуществлять межмодульное взаимодействие, в том числе и вызов методов. Для вызова методов аргументы записываются в определенные узлы базы. Вызов метода происходит по событию чтения из узла, зарезервированного под результат вызова. После обработки события результат оказывается в нужном узле.

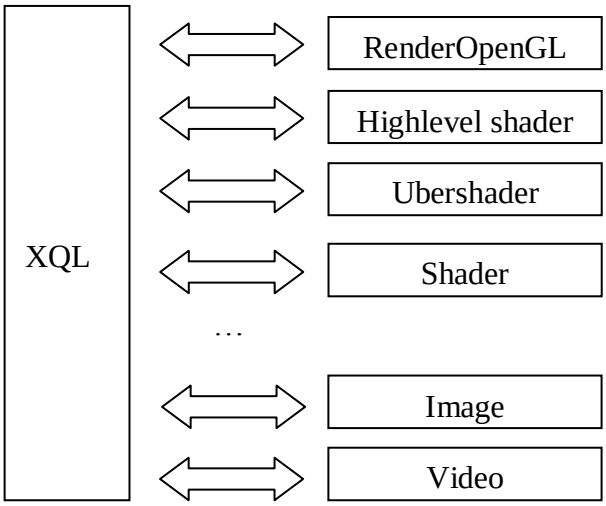

Рисунок 4: Схема межмодульного взаимодействия

Модулю высокоуровневых шейдеров подаются на вход два адреса: адрес шейдера поверхности и адрес шейдера модели освещения. Сами высокоуровневые шейдеры хранятся в XQL-базе в следующем виде:

```
SURFACE.surfaceShaderName =
{
  CODE  = "код шейдера"
  BOOL_DEFS = "перечень флагов, управляющих компиляцией"
  INT_DEFS = "перечень целочисленных констант, управляющих компиляцией"
}
```

Аналогично задаются шейдеры модели освещения, с той лишь разницей, что они хранятся в дочерних узлах узла `LIGHTING`.

При генерации код шейдеров считанный из нужных узлов подставляется в шаблон. Некоторые особенности реализации этого шаблона:

- Шаблон позволяет генерировать шейдер, поддерживающий произвольное количество источников света.
- Источники могут быть одного из трех типов: конусный, точечный и направленный. Однако два последних типа эмулируются через первый (это сделано для того, чтоб упростить код шаблона): точечный источник – это конус с углом раствора 360 °, а направленный – это точечный, удаленный на достаточно большое расстояние от освещаемого объекта в направлении противоположном направлению источника.
- Реализованы тени с помощью метода Shadow Maps. Поддерживаются два типа теней: жесткие и мягкие тени, в последнем случае используется размытие теней с помощью PCF (Percentage Closer Filtering) [9].
- Поддерживаются текстурные преобразования (поворот, масштаб, сдвиг). Для каждой текстуры можно задать свое преобразование.
- Прозрачность объекта можно задать как отдельной текстурой (opacity), так и в альфа-канале диффузной текстуры.
- Реализованы карты статического освещения (lightmap) .
- Поддерживаются отражения, используются заранее заготовленные текстуры окружения, наложение которых осуществляется сферическим проецированием.

Запрос на генерацию из высокоуровневых шейдеров осуществляется после загрузки материала, либо при смене его свойств SURFACE, LIGHTING. Сгенерированному убершейдеру назначается уникальное имя, которое строится на основе имен высокоуровневых шейдеров, из которых он был получен.

Модуль условной компиляции по заданному убершейдеру, значениям флагов и констант генерирует шейдер. Имена используемых в убершейдере флагов и констант перечисляются в аннотации убершейдера. Для более компактного хранения флаги и константы объединены в структуру:

```
struct UbershaderKey
{
     unsigned int boolDefs;
     unsigned int intDefs;
};
```

В ней под флаги и константы заведено по 32 бита. Таким образом, в убершейдере поддерживается до 32-х булевых флагов, и до 4-х целочисленных констант (по байту на каждую константу). Эта же самая структура используется при кэшировании шейдеров в качестве ключа поиска. Транслятор шейдеров из языка Cg в GLSL реализован с помощью

библиотеки Nvidia Cg Toolkit и представляет собой отдельный модуль (Shader). Одновременно с трансляцией происходит обработка шейдера препроцессором. Шейдер, полученный в результате трансляции, записывается в XQL-базу, где ему назначается уникальное имя, образуемое из имени убершейдера и представленных в виде строк значений флагов и констант. Генерация шейдера с помощью условной компиляции (либо взятие его из кеша) происходит непосредственно в момент визуализации объекта, поскольку на этой стадии уже определено, какие источники освещения влияют на него.

Для связи системы визуализации с модулем условной компиляции реализована система параметров, являющаяся частью системы визуализации. Эта же система помимо формирования флагов и констант для условной компиляции используется также для установки параметров в шейдеры. Параметры в системе имеют строковые имена и могут быть следующих типов: float, int, bool, вектор размерности от 2 до 4 с элементами типа int или float, матрица 4x4 с элементами типа float, текстура, массив с элементами любого из перечисленных типов. Для установки флагов и целочисленных констант используются типы bool и int. Сцена, материалы и элементы сцены предоставляют свои свойства через источники параметров (ParameterSource). Формирование входов модуля условной компиляции и установка параметров в шейдер осуществляется посредством приемников параметров (UbershaderDefineReceiver и ShaderParameterReciever). При этом приемнику не нужно иметь информацию, от какого источника был получен параметр, достаточно только имени, под которым этот параметр хранится в системе.

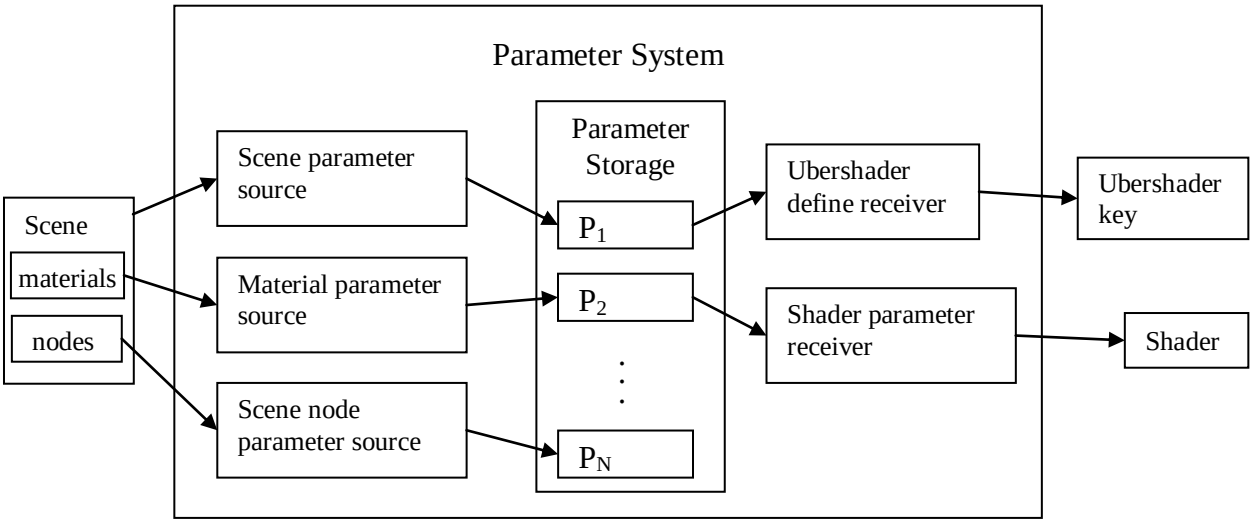

Рисунок 5: Система параметров

На данный момент система автоматизированной генерации шейдеров используется в системе визуализации, реализованной с помощью OpenGL. Благодаря тому, что в качестве языка шейдеров был выбран Cg, систему генерации шейдеров можно применить и для других систем визуализации, использующих DirectX 9 – 11.

3.2 Применение генерации шейдеров

3.2.1 Шейдер стандартного материала

С помощью системы автоматизированной генерации шейдеров в систему визуализации был добавлен ряд новых возможностей. В первую очередь, это стандартный материал. Шейдер стандартного материала генерируется из стандартного шейдера поверхности и шейдера модели освещения по Фонгу. Стандартный шейдер поверхности позволяет задать с помощью текстур диффузный и бликовый цвет поверхности, цвет самосвечения, карту нормалей для расчета микрорельефа. При визуализации микрорельефа используются вычисленные на центральном процессоре тангенты и бинормали, хранящиеся в вершинных атрибутах. Так же реализован альтернативный способ расчета микрорельефа, в котором тангенты и бинормали вычисляются в шейдере с помощью операций взятия частных производных [10]. Этот способ позволяет экономить видеопамять, но для его использования необходима видеокарта, поддерживающая модель пиксельных шейдеров 3.0. Стандартный материал, как и все остальные материалы в системе, поддерживает освещение от произвольного количества источников света с учетом падающих теней. В стандартном материале можно задать до 8-ми текстур: 4 из них (diffuse, specular, emission, bump) используются в шейдере поверхности, остальные 4 (ambient, opacity, lightmap, reflection) изначально присутствуют в шаблоне, по которому происходит генерация шейдеров.

3.2.2 Трипланарное текстурирование

В качестве примера использования шейдеров поверхности было реализовано трипланарное текстурирование [11]. Стандартный способ текстурирования предполагает, что в вершинах объекта хранятся текстурные координаты, которые, как правило, настраиваются в редакторе трехмерного моделирования. Задать текстурные координаты (текстурную развертку) для сложного невыпуклого объекта так, чтобы текстура отображалась без растяжений и швов, весьма нетривиальная задача. Для аналогии можно представить, что требуется обернуть трехмерный объект прямоугольным листом бумаги без складок и разрывов.

Трипланарное текстурирование позволяет равномерно затекстурировать объект, не используя текстурную развертку. При расчете цвета производятся три выборки из текстуры, где в качестве текстурных координат берется проекция позиции обрабатываемой точки на одну из плоскостей (XOY, XOZ, YOZ). Затем, три считанных значения цвета смешиваются с использованием компонент нормали как весовых коэффициентов. Код шейдера, реализующего трипланарное текстурирование, представлен

в Приложении А. Данный тип текстурирования хорошо подходит для наложение текстур с регулярным рисунком, таких как камень или дерево, на объекты сложной формы. Так же этот способ применим для ландшафтов.

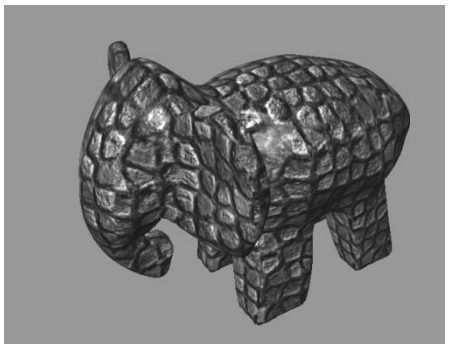

Рисунок 6: Трипланарное текстурирование

3.2.3 Модель освещения Кука-Торренса

Помимо стандартного шейдера модели освещения, рассчитывающего освещение по Фонгу, был реализован более сложный способ расчета освещения – модель Кука-Торренса [12]. В этой модели используются физически обоснованные формулы, и она хорошо подходит для визуализации металлических поверхностей.

Будем использовать следующие обозначения:

N – нормаль поверхности

L – вектор, противоположный направлению луча света

V – вектор, противоположный направлению взгляда

H – нормализованная сумма векторов V и L

Сила диффузной составляющей вычисляется по стандартной формуле:

$$K_d = \vec{N} \cdot \vec{L}$$

Сила зеркальной составляющей зависит от трех факторов:

- Коэффициент Френеля (F)
- Геометрическая составляющая, учитывающая самозатенение (G)
- Компонент, учитывающий шероховатость поверхности (D)

Общая формула для вычисления силы зеркальной составляющей такова:

$$K_s = \frac{F \cdot G \cdot D}{(\vec{V} \cdot \vec{N}) \cdot (\vec{L} \cdot \vec{N})}$$

Геометрическая составляющая рассчитывается следующим образом:

$$G = \min(1, \frac{2(\vec{H} \cdot \vec{N})(\vec{V} \cdot \vec{N})}{(\vec{V} \cdot \vec{H})}, \frac{2(\vec{H} \cdot \vec{N})(\vec{L} \cdot \vec{N})}{(\vec{V} \cdot \vec{H})})$$

Для вычисления компонента, учитывающего шероховатость поверхности, используется распределение Бекмана:

$$D = \frac{1}{4m^2(\overrightarrow{H} \cdot \vec{N})^4} \cdot e^{\left(\frac{(\vec{H}\cdot\vec{N})^2 - 1}{m^2(\vec{H}\cdot\vec{N})^2}\right)}$$

где параметр m (от 0 до 1) определяет шероховатость поверхности. Чем он больше, тем поверхность шероховатее, а следовательно, отражает свет под более широкими углами.

Для вычисления коэффициента Френеля применяется аппроксимация Шлика:

$$F = F_0 + (1 - (\vec{V} \cdot \vec{N}))^5 \cdot (1 - F_0)$$

где F_0 – количество отраженного света при нормальном падении (перпендикулярно поверхности).

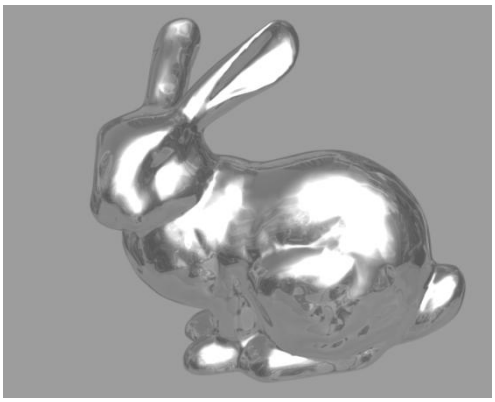

Рисунок 7: Модель освещения Кука-Торрентса

Код шейдера модели освещения Кука-Торренса представлен в Приложении Б.

Заключение

Были получены следующие результаты:

1. Изучены и классифицированы подходы, применяемые для генерации шейдеров.
2. Разработан формат описания материалов на базе языка XQL.
3. Разработан комбинированный способ генерации шейдеров, строящий шейдеры из высокоуровневых шейдеров (шейдера поверхности и шейдера модели освещения) с использованием условной компиляции.
4. Реализована система автоматизированной генерации шейдеров. Применяемые в ней подходы обеспечивают генерацию высокопроизводительных шейдеров. Возможности системы могут быть расширены путем написания высокоуровневых шейдеров.
5. Условная компиляция позволила повысить производительность визуализации в 1.5 – 2 раза.
6. Система генерации шейдеров была интегрирована с системой визуализации, реализованной на основе OpenGL.
7. С помощью высокоуровневых шейдеров были реализованы стандартный материал, трипланарное текстурирование, модель освещения Кука-Торренса.

Дальнейшие направления работы:

1. Разработка способов процедурной генерации текстур с помощью шейдеров поверхности.
2. Реализация дополнительных моделей освещения, включая анизотропные.
3. Реализация возможности сохранение кеша сгенерированных шейдеров на жесткий диск с целью уменьшения времени загрузки приложения.

Литература

1. Долговесов Б.С., 3D графика реального времени: от тренажеров до виртуальных студий // Graphicon 2005, Новосибирск
2. Akenine-Moller T. Real-time rendering. 2008.
3. Kilgard. M The Cg tutorial. 2003
4. Сладков Д. Промышленные убершейдеры. // Труды конференции КРИ-2008. 2008.
5. http://unity3d.com/learn/documentation
6. http://www.unrealengine.com/udk/documentation/
7. Долговесов Б. С. Объектно-ориентированная база данных в интерактивных системах 3d визуализации. // Вестник Новосибирского Государственного Университета. 2011.
8. Gamma E., Helm R., Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. 1994
9. John R. IsidoroShadow Mapping: GPU-based Tips and Techniques. 2006
10. Артиков А. Н., Артиков Т. Н. Визуализация микрорельефа на анимированных трехмерных объектах. // Труды конференции МНСК 2012
11. Nicholson K. GPU Based Algorithms for Terrain Texturing. 2008.
12. Cook R., Torrance K. A Reflectance Model for Computer Graphics. 1982.

Приложение А
(обязательное)
Шейдер поверхности для трипланарного текстурирования

```
SurfaceOutput surface(SurfaceInput input)
{
   SurfaceOutput output;
   float3 objectNormal = mul(float4(input.normal, 0), mWorldInverse).xyz;
   float3 k = abs(normalize(objectNormal));
   k = pow(max(float3(0.0, 0.0, 0.0), k - float3(0.2, 0.2, 0.2)), 10);
   k /= (k.x + k.y + k.z);

 #ifdef MAP__BUMP__EXIST
   float3 bumpTc = transform(input.objectPosition, bumpTcTransform);
   float2 b1 = 2 * tex2D(bumpSampler, bumpTc.yz).xy - float2(1, 1);
   float2 b2 = 2 * tex2D(bumpSampler, bumpTc.zx).xy - float2(1, 1);
   float2 b3 = 2 * tex2D(bumpSampler, bumpTc.xy).xy - float2(1, 1);
   float3 bumpNormal = float3(0, b1.x, b1.y)*k.x +
                       float3(b2.y, 0, b2.x)*k.y +
                       float3(b3.x, b3.y, 0)*k.z;
   output.normal = normalize(input.normal + bumpWeight * bumpNormal);
 #else
   output.normal = input.normal;
 #endif

 #ifdef MAP__DIFFUSE__EXIST
   float3 diffuseTc = transform(input.objectPosition, diffuseTcTransform);
   float4 d1 = tex2D(diffuseSampler, diffuseTc.yz);
   float4 d2 = tex2D(diffuseSampler, diffuseTc.zx);
   float4 d3 = tex2D(diffuseSampler, diffuseTc.xy);
   output.diffuse = d1*k.x + d2*k.y + d3*k.z;
   output.diffuse = lerp(diffuseColor, output.diffuse, diffuseWeight);
 #else
   output.diffuse = diffuseColor;
 #endif

 #ifdef MAP__SPECULAR__EXIST
   float3 specularTc = transform(input.objectPosition, specularTcTransform);
   float4 s1 = tex2D(specularSampler, specularTc.yz);
   float4 s2 = tex2D(specularSampler, specularTc.zx);
   float4 s3 = tex2D(specularSampler, specularTc.xy);
   output.specular = s1*k.x + s2*k.y + s3*k.z;
   output.specular = lerp(specularColor, output.specular, specularWeight);
#else
   output.specular = specularColor;
#endif

   output.emission = float4(0, 0, 0, 0);
   return output;
}
```

Приложение Б
(обязательное)
Шейдер модели освещения Кука-Торренса

```
float l_roughness : ROUGHNESS = 0.5;
float l_F0 : F0 = 1.3;

LightingOutput lighting(LightingInput input)
{
    LightingOutput output;
    float3  H = -normalize(input.lightDir + input.viewDir);
    float NdotL = max(0, dot(input.normal, -input.lightDir));
    float NdotV = max(0, dot(input.normal, -input.viewDir));
    float NdotH = max(1.0e-7, dot(input.normal, H));
    float VdotH = max(0, dot(-input.viewDir, H));

    float r_sq = l_roughness * l_roughness;
    float NdotH_sq = NdotH * NdotH;
    float D = 1 / (4 * r_sq * NdotH_sq * NdotH_sq) *
              exp((NdotH_sq - 1.0) / (r_sq * NdotH_sq));

    float G1 = 2 * NdotH * NdotV / VdotH;
    float G2 = 2 * NdotH * NdotL / VdotH;
    float G = min(1.0, min(G1, G2));

    float F = l_F0 + (1.0 - l_F0) * pow (1.0 - NdotV, 5.0f);

    output.diffuseK = NdotL;
    output.specularK =  NdotL * F * D * G / (NdotL * NdotV + 1.0e-7);
    return output;
}
```