МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ОБНИНСКИЙ ИНСТИТУТ АТОМНОЙ ЭНЕРГЕТИКИ — филиал
федерального государственного автономного образовательного учреждения
высшего профессионального образования
«Национальный исследовательский ядерный университет «МИФИ»
(ИАТЭ НИЯУ МИФИ)

Факультет кибернетики
Кафедра компьютерных систем, сетей и технологий

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К УЧЕБНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

Исследование графического движка OptiX

Студент гр. ВТ-С10 ..... Еличева Е.А.

Руководитель ..... Белявцев И.П.

Обнинск 2014

Содержание

Введение

Трассировка лучей (англ. Ray tracing; рейтрейсинг) — один из методов геометрической оптики — исследование оптических систем путём отслеживания взаимодействия отдельных лучей с поверхностями. В узком смысле — технология построения изображения трёхмерных моделей в компьютерных программах, при которых отслеживается обратная траектория распространения луча (от экрана к источнику). Данный метод имеет следующие достоинства:

1. возможность рендеринга гладких объектов без аппроксимации их полигональными поверхностями (например, треугольниками);
2. вычислительная сложность метода слабо зависит от сложности сцены;
3. высокая алгоритмическая распараллеливаемость вычислений — можно параллельно и независимо трассировать два и более лучей, разделять участки (зоны экрана) для трассирования на разных узлах кластера и т.д;
4. отсечение невидимых поверхностей, перспектива и корректное изменения поля зрения являются логическим следствием алгоритма.

Серьёзным недостатком метода обратного трассирования является производительность. Метод растеризации и сканирования строк использует когерентность данных, чтобы распределить вычисления между пикселями. В то время как метод трассирования лучей каждый раз начинает процесс определения цвета пикселя заново, рассматривая каждый луч наблюдения в отдельности. Впрочем, это разделение влечёт появление некоторых других преимуществ, таких как возможность трассировать больше лучей, чем предполагалось для устранения контурных неровностей в определённых местах модели. Также это регулирует отражение лучей и эффекты преломления, и в целом — степень фотореалистичности изображения. [4]

Для трассировки лучей NVIDIA предлагает программную библиотеку Optix, позволяющую разработчикам программного обеспечения быстро создавать приложения на основе трассировки лучей и быстро достигать результатов благодаря графическим процессорам NVIDIA и традиционным программам на языке C. В отличие от рендерера с неизменяемым внешним видом, ограниченного определенными структурами данных или поддерживаемым языком программирования, движок OptiX носит чрезвычайно общий характер, позволяя разработчикам программного обеспечения быстро ускорять выполнение любых задач на основе трассировки лучей и выполнять их на широко доступном оборудовании.

Целью данной учебно–исследовательской работы является создание демонстрационного приложениея использованием графического движка OptiX.

Задачи, решаемые в ходе работы:

1. Изучение программно-аппаратной архитектуры CUDA
2. Изучение принципов функционирования графического движка OptiX
3. Изучение процедуры установки графического движка OptiX

4. Изучение встроенных примеров графического движка OptiX

5. Разработка демонстрационного приложения

6. Выяснение перспектив применимости графического движка в прикладных приложениях

1 Изучение программно-аппаратной архитектуры CUDA

1.1 GPGPU

CUDA (Compute Unified Device Architecture) – это технология от компании NVidia, предназначенная для разработки приложений для массивно-параллельных вычислительных устройств (в первую очередь для GPU начиная с серии G80).

Основными плюсами CUDA являются ее бесплатность (SDK для всех основных платформ свободно скачивается с developer.nvidia.com), простота (программирование ведется на "расширенном C") и гибкость.

Фактически CUDA является дальнейшим развитием GPGPU (General Purpose computation on GPU). Дело в том, что уже с самого начала GPU активно использовали параллельность (как вершины, так и отдельные фрагменты могут обрабатываться параллельно и независимо друг от друга, т.е. очень хорошо ложатся на параллельную архитектуру).

По мере развития GPU росла как степень распараллеливания, так и гибкость самих GPU. Самые первые GPU для PC (Voodoo) фактически представляли собой просто растеризатор с возможностью наложения текстуры и буфером глубины. Довольно быстро появились GPU с T&L, т.е. полной обработкой вершин на самом GPU - на вход поступают трехмерные данные и на выходе получаем готовое изображение (например, Riva TNT). Однако гибкость в них была небольшой - ведь все вычисления велись в рамках фиксированного конвейера (FFP).

Следующим шагом (GeForce 2) стало появления вершинных шейдеров (расширение ARB_vertex_program) - обработку вершин стало возможным задавать в виде программы, написанной на специальном ассемблере. При этом вершины обрабатывались параллельно и независимо друг от друга. Ниже приводится пример простой программы на таком ассемблере.

```
!!ARBvp1.0
ATTRIB pos     = vertex.position;
PARAM  mat [4] = { state.matrix.mvp };

# transform by concatenation of modelview and projection matrices

DP4 result.position.x, mat [0], pos;
DP4 result.position.y, mat [1], pos;
DP4 result.position.z, mat [2], pos;
DP4 result.position.w, mat [3], pos;
```

```
# copy primary color

MOV result.color, vertex.color;
END
```

Вполне логичным следующим шагом стало появление фрагментных программ (расширение ARB_fragment_program), позволяющим задавать расчет каждого пиксела также при помощи программы, на ассемблере. Важным моментом является то, что все эти вычисления (как для вершин, так и для фрагментов) ведутся с использованием 32-битовых floating-point чисел.

В архитектуре GPU появились отдельные вершинные и фрагментные процессоры, выполняющие соответствующие программы. Данные процессоры вначале были крайне просты - можно было выполнять лишь простейшие операции, практически полностью отсутствовало ветвление и все процессоры одного типа одновременно выполняли одну и ту же команду (классическая SIMD-архитектура).

За счет большого числа вершинных и фрагментных процессоров, выполняющих такие программы, оказалось, что по быстродействию (измеряемому в количестве floating-point операций в секунду) GPU в разы обгоняют CPU.

Заключительным шагом, превратившим GPU в мощные параллельные вычислители, стало поддержка floating-point текстур, т.е. стало возможным хранить значения в текстурах как 32-битовые floating-point числа.

В результате GPU фактически стало устройством, реализующим потоковую вычислительную модель (stream computing model) - есть потоки входных и выходных данных, состоящие из одинаковых элементов, которые могут быть обработаны независимо друг от друга. Обработка элементов осуществляется ядром (kernel) (см. рис. 1).

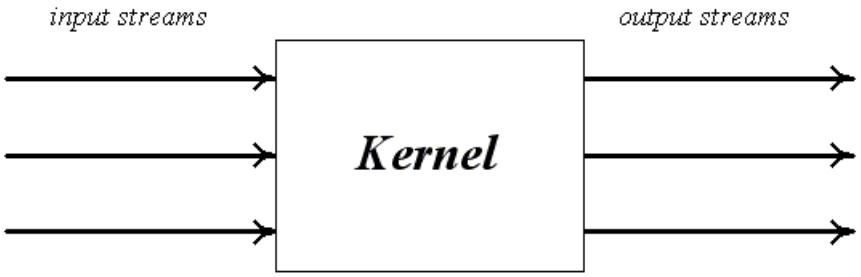

Рисунок 1 – Потоковые вычисления

Фактически GPU оказалось мощным SIMD (Single Instruction Multiple Data) процессором. В результате появилось GPGPU - использование огромной вычислительной мощности GPU для решения неграфических задач. Несмотря на значительные результаты GPGPU обладало рядом недостатков:

1. вся работа шла через графический API, код для GPU писался на GLSL/HLSL/Cg, остальной код - на традиционном языке программирования

2. наличие ограничений на размеры и размерность текстур;

3. полностью отсутствовала возможность взаимодействия между параллельно обрабатываемыми пикселами;

4. отсутствовала поддержка так называемого scatter'а (хотя были найдены обходные пути);

Кроме того на GPU GeForce 6xxx/7xxx отсутствовала нативная поддержка целых чисел и побитовых операций над ними.

Появление CUDA (а также GPU G80) полностью сняло все эти ограничения, предложив для GPGPU простую и удобную модель. В этой модели GPU рассматривается как специализированное вычислительное устройство (называемое device), которое:

1. является сопроцессором к CPU (называемому host)

2. обладает собственной памятью (DRAM)

3. обладает возможностью параллельного выполнения огромного количества отдельных нитей (threads)

При этом между нитями на CPU и нитями на GPU есть принципиальные различия —

1. нити на GPU обладают крайне "небольшой стоимостью их создание и управление требует минимальных ресурсов (в отличии от CPU)

2. для эффективной утилизации возможностей GPU нужно использовать многие тысячи отдельных нитей (для CPU обычно нужно не более 10-20 нитей)

Сами программы пишутся на "расширенном"C, при этом их параллельная часть (ядра) выполняется на GPU, а обычная часть - на CPU. CUDA автоматически осуществляет разделением частей и управлением их запуском.

CUDA использует большое число отдельных нитей для вычислений, часто каждому вычисляемому элементами соответствует одна нить. Все нити группируются в иерархию - grid/block/thread (см. рис. 2).

Верхний уровень - grid - соответствует ядру и объединяет все нити выполняющие данное ядро. grid представляет собой одномерный или двухмерный массив блоков (block). Каждый блок (block) представляет из себя одно/двух/трехмерный массив нитей (threads).

При этом каждый блок представляет собой полностью независимый набор взаимодействующих между собой нитей, нити из разных блоков не могут между собой взаимодействовать.

Фактически блок соответствует независимо решаемой подзадаче, так например если нужно найти произведение двух матриц, то матрицу-результат можно разбить на отдельные подматрицы одинакового размера. Нахождение каждой такой подматрицы может происходить абсолютно независимо от нахождения остальных подматриц. Нахождение такой подматрицы - задача отдельного блока, внутри блока каждому элементу подматрицы соответствует отдельная нить.

При этом нити внутри блока могут взаимодействовать между собой (т.е. совместно решать подзадачу) через

1. общую память (shared memory)

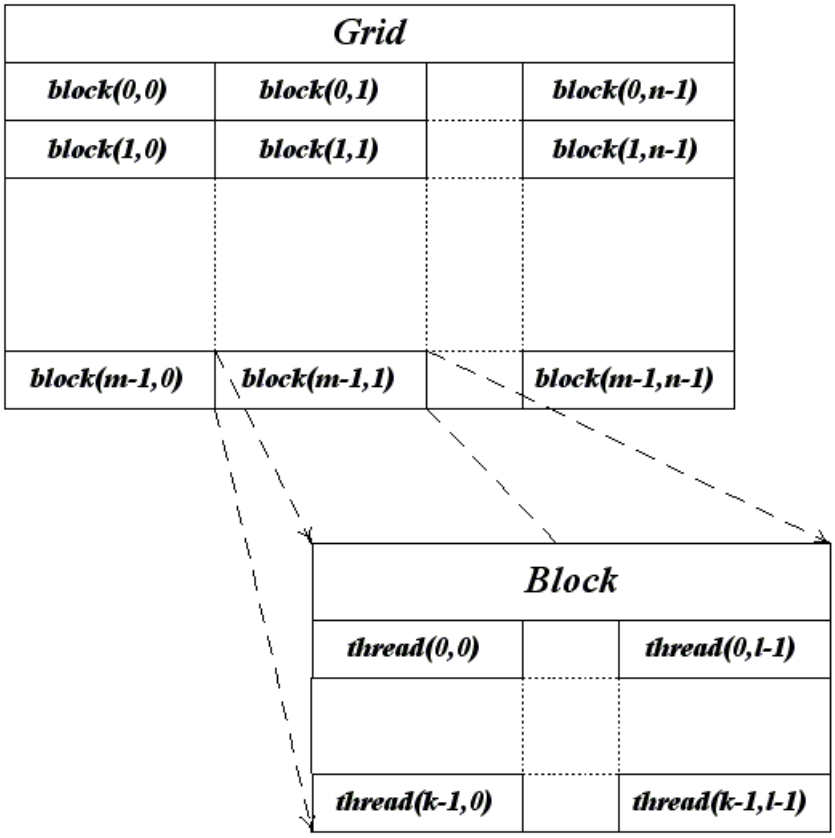

Рисунок 2 – Иерархия нитей в CUDA

2. функцию синхронизации всех нитей блока (__synchronize)

Подобная иерархия довольно естественна - с одной стороны хочется иметь возможность взаимодействия между отдельными нитями, а с другой - чем больше таких нитей, тем выше оказывается цена подобного взаимодействия.

Поэтому исходная задача (применение ядра к входным данным) разбивается на ряд подзадач, каждая из которых решается абсолютно независимо (т.е. никакого взаимодействия между подзадачами нет) и в произвольном порядке.

Сама же подзадача решается при помощи набора взаимодействующих между собой нитей.

С аппаратной точки зрения все нити разбиваются на так называемые warp'ы — блоки подряд идущих нитей, которые одновременно (физически) выполняются и могут взаимодействовать друг с другом. Каждый блок нитей разбивается на несколько warp'ов, размер warp'а для всех существующих сейчас GPU равен 32.

Важным моментом является то, что нити фактически выполняют одну и ту же команды, но каждая со своими данными. Поэтому если внутри warp'а происходит ветвление (например в результате выполнения оператора if), то все нити warp'а выполняют все возникающие при этом ветви. Поэтому крайне желательно уменьшить ветвление в пределах каждого отдельного warp'а.

Также используется понятие half-warp'а — это первая или вторая половина warp'а. Подобное разбиение warp'а на половины связано с тем, что обычно обращение к памяти делаются отдельно для каждого half-warp'а.

Кроме иерархии нитей существует также несколько различных типов памяти. Быстродействие приложения очень сильно зависит от скорости работы с памятью. Именно поэтому в традиционных CPU большую часть кристалла занимают различные кэши, предназначенные для ускорения работы с памятью (в то время как для GPU основную часть кристалла занимают ALU).

В CUDA для GPU существует несколько различных типов памяти, доступных нитям, сильно различающихся между собой (см. табл. 2).

Таблица 1 – Типы памяти в CUDA.

Тип памяти	Доступ	Уровень выделения	Скорость работы
регистры (registers)	R/W	per-thread	высокая (on chip)
local	R/W	per-thread	низкая (DRAM)
shared	R/W	per-block	высокая (on-chip)
global	R/W	per-grid	низкая(DRAM)
constant	R/O	per-grid	высокая(on chip L1 cache)
texture	R/O	per-grid	высокая(on chip L1 cache)

При этом CPU имеет R/W доступ только к глобальной, константной и текстурной памяти (находящейся в DRAM GPU) и только через функции копирования памяти между CPU и GPU (предоставляемые CUDA API).

1.2 Расширения языка C

Программы для CUDA (соответствующие файлы обычно имеют расширение .cu) пишутся на "расширенном"C и компилируются при помощи команды nvcc.

Вводимые в CUDA расширения языка C состоят из:

1. спецификаторов функций, показывающих где будет выполняться функция и откуда она может быть вызвана;
2. спецификаторы переменных, задающие тип памяти, используемый для данной переменных;
3. директива, служащая для запуска ядра, задающая как данные, так и иерархию нитей;
4. встроенные переменные, содержащие информацию о текущей нити;
5. runtime, включающий в себя дополнительные типы данных;

Таблица 2 – Спецификаторы функций в CUDA.

Спецификатор	Выполняется на	Может вызываться из
__device__	device	device
__global__	device	host
__hos__	host	host

1.2.1 Спецификаторы

При этом спецификаторы __host__ и __device__ могут быть использованы вместе (это значит, что соответствующая функция может выполняться как на GPU, так и на CPU — соответствующий код для обеих платформ будет автоматически сгенерирован компилятором). Спецификаторы __global__ и __host__ не могут быть использованы вместе.

Спецификатор __global__ обозначает ядро и соответствующая функция должна возвращать значение типа void.

```
__global__ void myKernel ( float * a, float * b, float * c )
{
    int index = threadIdx.x;

    c [i] = a [i] * b [i];
}
```

На функции, выполняемые на GPU (__device__ и __global__) накладываются следующие ограничения:

1. нельзя брать их адрес (за исключением __global__ функций);
2. не поддерживается рекурсия;
3. не поддерживаются static-переменные внутри функции;
4. не поддерживается переменное число входных аргументов;

Для задания размещения в памяти GPU переменных используются следующие спецификаторы — __device__, __constant__ и __shared__. На их использование также накладывается ряд ограничений:

1. эти спецификаторы не могут быть применены к полям структуры (struct или union);
2. соответствующие переменные могут использоваться только в пределах одного файла, их нельзя объявлять как extern;
3. запись в переменные типа __constant__ может осуществляться только CPU при помощи специальных функций;
4. __shared__ переменные не могут инициализироваться при объявлении;

1.2.2 Добавленные переменные

В язык добавляются 1/2/3/4–мерные вектора из базовых типов – char1, char2, char3, char4, uchar1, uchar2, uchar3, uchar4, short1, short2, short3, short4, ushort1, ushort2, ushort3, ushort4, int1, int2, int3, int4, uint1, uint2, uint3, uint4, long1, long2, long3, long4, ulong1, ulong2, ulong3, ulong4, float1, float2, float3, float2, и double2.

Обращение к компонентам вектора идет по именам — x, y, z и w. Для создания значений — векторов заданного типа служит конструкция вида make_<typeName>.

```
int2   a = make_int2   ( 1, 7 );
float3 u = make_float3 ( 1, 2, 3.4f );
```

Обратите внимание, что для этих типов (в отличии от шейдерных языков GLSL/Cg/HLSL) не поддерживаются векторные покомпонентные операции, т.е. нельзя просто сложить два вектора при помощи оператора "+— это необходимо явно делать для каждой компоненты.

Также для задания размерности служит тип dim3, основанный на типе uint3, но обладающий нормальным конструктором, инициализирующим все не заданные компоненты единицами.

1.2.3 Директива вызова ядра

Для запуска ядра на GPU используется следующая конструкция:

```
kernelName <<<Dg,Db,Ns,S>>> ( args )
```

Здесь kernelName это имя (адрес) соответствующей __global__ функции, Dg — переменная (или значение) типа dim3, задающая размерность и размер grid'a (в блоках), Db — переменная (или значение) типа dim3, задающая размерность и размер блока (в нитях), Ns — переменная (или значение) типа size_t, задающая дополнительный объем shared–памяти, которая должна быть динамически выделена (к уже статически выделенной shared-памяти), S – переменная (или значение) типа cudaStream_t задает поток (CUDA stream), в котором должен произойти вызов, по умолчанию используется поток 0. Через args обозначены аргументы вызова функции kernelName.

Также в язык C добавлена функция __syncthreads, осуществляющая синхронизацию всех нитей блока. Управление из нее будет возвращено только тогда, когда все нити данного блока вызовут эту функцию. Т.е. когда весь код, идущий перед этим вызовом, уже выполнен (и, значит, на его результаты можно смело рассчитывать). Эта функция очень удобная для организации безконфликтной работы с shared-памятью.

Также CUDA поддерживает все математические функции из стандартной библиотеки C, однако с точки зрения быстродействия лучше использовать их float–аналоги (a не double) — например sinf. Кроме этого CUDA предоставляет дополнительный набор математических функций (__sinf, __powf и т.д.) обеспечивающие более низкую точность, но заметно более высокое быстродействие чем sinf, powf и т.п. [2]

2 Optix: движок трассировки лучами общего назначения

Рисунок 3 – Изображение из приложения созданого в OptiX.

2.1 Краткий обзор

Механизм трассировки лучей NVIDIA® OptiX™ — программируемая система, разработанная для графических ускорителей NVIDIA и других массивно-параллельных архитектур. Механизм OptiX основывается на базовом наблюдении, что большинство алгоритмов трассировки лучей могут быть реализованы, используя маленький набор программируемых операций. Следовательно, ядро OptiX — проблемно-ориентированный JIT компилятор, который генерирует пользовательские ядра трассировки лучей, комбинируя предоставленные пользователями программы для генерации луча, заливки материала, объектного пересечения и обхода сцены. Оно включает реализацию очень широкого набора основанных на трассировке лучей алгоритмов и приложений, включая интерактивный рендеринг, оффлайн рендеринг, системы обнаружения коллизий, запросы искусственного интеллекта и научного моделирования, такие как звуковое распространение. OptiX достигает высокой производительности через компактную объектную модель и применения нескольких специфичной для трассировки лучей оптимизаций компилятора. Для простоты использования OptiX представляет модель программирования единственного луча с полной поддержкой рекурсии и динамического механизма отправки, подобного вызовам виртуальной функции.

2.2 Введение

Чтобы решить проблему создания доступной, гибкой, и эффективной системы трассировки лучей для массивно-параллельной архитектуры, представляем OptiX — механизм трассировки лучей общего назначения. Этот механизм комбинирует программируемый конвейер трассировки лучей с легким представлением сцены. Общий интерфейс программирования включает реализацию множества основанных на трассировке лучей алгоритмов в графических и нерафических доменах, таких как рендеринг, звуковое распространение, обнаружение коллизий и искусственный интеллект.

В этом разделе мы обсуждаем цели проекта OptiX, а также реализации для NVIDIA Quadro®, GeForce® и Tesla® GPUs. Мы рассмотрим проблемно-ориентированную компиляцию с гиб-

ким набором средств управления иерархией сцены, ускоряющего создания структуры и обхода, непрерывного обновления сцены и динамично сбалансированной с загрузки модели выполнения GPU.

Чтобы создать систему для широкого диапазона задач трассировки лучей, были принято несколько компромиссов и проектных решений, которые привели к следующим особенностям:

- Общий низкоуровневый механизм трассировки лучей. Механизм OptiX фокусируется исключительно на фундаментальных вычислениях, требуемых для трассировки лучей, и избегает встраивания конструкции для рендеринга. Движок представляет механизмы для выражения взаимодействий геометрии луча и не имеет встроенного понятия световых сигналов, теней, коэффициента отражения, и т.д.

- Программируемый конвейер трассировки лучей. Механизм OptiX демонстрирует, что большинство алгоритмов трассировки лучей может быть реализовано, используя маленький набор легких программируемых операций. Это определяет абстрактную модель выполнения трассировки лучей, поскольку последовательность пользователя определила программы. Эта модель при объединении с произвольными данными, хранившимися с каждым лучом, может использоваться, чтобы реализовать множество сложных графических сцен и невизуальных алгоритмов.

- Простая модель программирования. Механизм OptiX обеспечивает такие механизмы выполнения, что программисты пользуются знакомыми методами работы с трассировкой лучами и не обременяют себя низкоуровневой оптимизацией. Движок представляет знакомую рекурсивную, модель программирования единственного луча, а не пакеты луча или явные конструкции SIMD-стиля.

- Проблемно-ориентированный компилятор. Механизм OptiX комбинирует своевременные методы компиляции со специфичным для трассировки лучей знанием, чтобы реализовать его модель программирования эффективно. Абстракция механизма разрешает компилятору настраивать модель выполнения для доступных системных аппаратных средств.

- Эффективное представление сцены. Механизм OptiX реализует объектную модель, которая использует динамическое наследование, чтобы упростить компактное представление параметров сцены. Гибкая система графика узла позволяет сцене быть организованной для максимальной производительности, все еще поддерживая инстанцирование, многоуровневую детализацию и вложенные ускоряющие структуры.

2.3 Программируемый конвейер трассировки лучей

Центральная идея механизма OptiX состоит в том, что большинство алгоритмов трассировки лучей может быть реализовано, используя маленький набор программируемых операций. Это прямой аналог к программируемым конвейерам растеризации, используемым OpenGL и Direct3D. На высоком уровне те системы представляют абстрактный растеризатор, содержащий

легкие обратные вызовы для штриховки вершины, обработки геометрии, составления мозаики и операций штриховки фрагмента. Ансамбль этих типов программ, обычно используемых в многократных передачах, может использоваться, чтобы реализовать широкий спектр основанных на растеризации алгоритмов.

Мы идентифицировали соответствующее абстрактное выполнение трассировки лучей модель вместе с легкими операциями, которые могут быть настроены реализовывать большое разнообразие основанных на трассировке лучей алгоритмов.[NVIDIA 2010a]. Эти операции или программы, могут быть объединенны с определяемой пользователем структурой данных (payload), связанной с каждым лучом. Ансамбль программ действует сообща для реализации алгоритма определенного клиентского приложения.

2.3.1 Программы

Есть семь различных типов программ в OptiX, каждая из которых работает над одним лучом одновременно. Кроме того, программа ограничительной рамки действует на геометрию, чтобы определить границы примитива для построения ускоряющей структуры. Сочетание пользовательских программ и жестко закодированного OptiX ядра образует конвейер трассировки лучей, который описан на рисунке 4. В отличие от конвейера прямой подачи растеризации более естественно думать о конвейере трассировки лучей как о графе вызовов. Основная операция, rtTrace, чередуется между поиском пересечения (Traverse) и реагированием на этой пересечение (Shade). При записи или чтении данных в объявленном пользователем payload луча и в массивах глобальной памяти устройства (буферы), эти операции объединяются для выполнения произвольных вычислений во время трассировки лучей.

Программы генерации лучей являются начальной точкой в конвейере трассировки лучей. Один вызов rtContextLaunch создаст много экземпляров этих программ. В примере на рисунке 3 программа генерации луча создает луч с использованием модели камеры pinhole для одного пикселя, начинает операцию трассировки и сохраняет результирующий цвет в выходной буфер. С помощью этого механизма, можно также выполнять другие операции, такие как создание фотонных карт, вычисления освещения методом отжига, обработки запросов лучей, переданных из OpenGL, стрельба несколькими лучами для суперсемплинга или реализации различных моделей камер. Программы пересечения реализуют тесты пересечения лучевой геометрии. При пересечении ускоряющих структур система будет вызывать программу пересечения, чтобы выполнить геометрический запрос. Программа определяет, и где луч касается объекта и может вычислять нормали, текстурных координат, или другие атрибуты в зависимости от положения попадания. Произвольное количество атрибутов могут быть связаны с каждым пересечением. Программы пересечения включают поддержку произвольных поверхностей, таких как сферы, цилиндры, поверхностей высокого порядка, или даже фрактальной геометрии, как множество Жюлиа. Тем не менее, даже в системах,состоящих только из треугольников, можно встретить большое разнообразие представлений сетки. Программируемая операция пересечения облегчает прямой доступ в оригинальном формате, который может помочь избежать копии при взаимодействии с системами на основе растеризации.

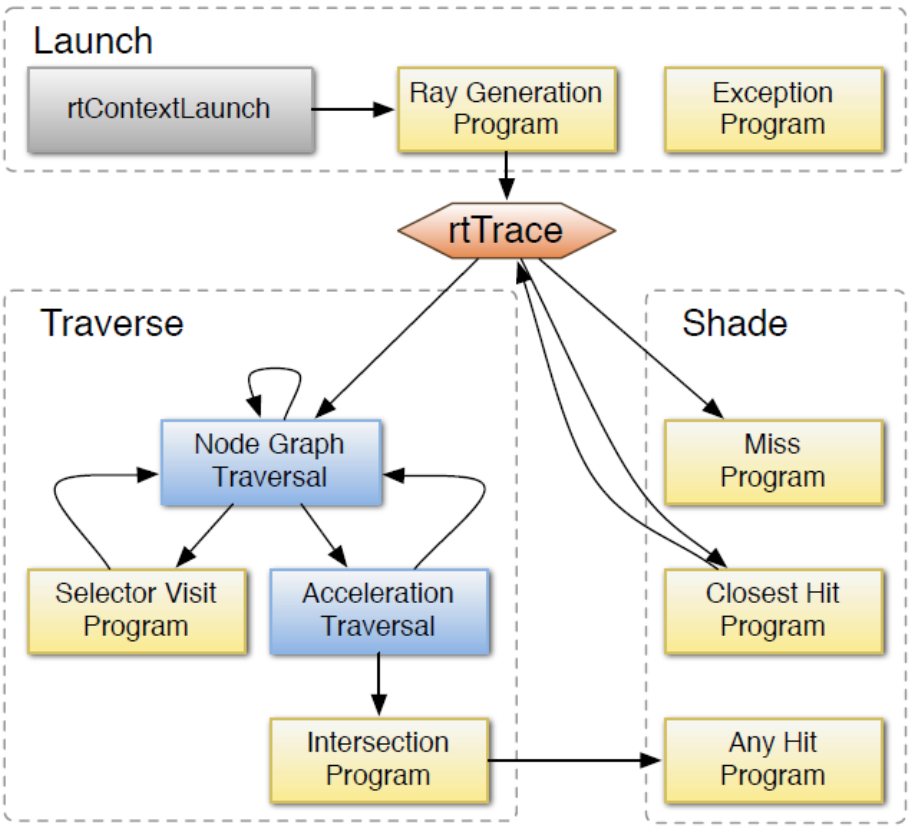

Рисунок 4 – Граф вызова, показывающий поток управления по конвейеру трассировки лучей.Желтые прямоугольники представляют указанные пользователем программы, а синие прямоугольники —алгоритмы внутренние для OptiX. Выполнение инициируется вызовом API rtContextLaunch.Встроенная функция, rtTrace, может быть использована в программе генерации лучей для отправки лучей в сцену. Эта функция также может вызываться рекурсивно программой ближайшего попадания для тени и вторичных лучей. Программа исключения выполняется, когда выполнение отдельного луча завершилось с ошибкой, такой как чрезмерное потребление памяти.

Программы ограничительной рамки вычисляют границы, связанные с каждым примитивом для включения ускоряющей структуры над произвольной геометрией. Учитывая примитивный индекс, простая программа такого типа может, например, читать вершинные данные из буфера и вычислять треугольники ограничительной рамки. Процедурная геометрия может иногда только оценить границы примитива. Такие оценки допускаются пока они консервативны, но свободные границы могут привести к снижению производительности.

Ближайшие программы попадания вызываются один раз обхода нашла ближайший пересечение луча с геометрии сцены. Этот тип программы очень напоминает поверхностный шейдеров в классических системах визуализации. Как правило, программа ближайшего попадания будет выполнять вычисления, такие как затенение, потенциальное излучение новых лучей в процессе, и сохранение результирующих данных в payload луча.

Программы любого попадания вызываются во время обхода для каждого пересечения объек-

та луча, который находится. Программа любого попадания позволяет материалу участвовать в решении объектного пересечения, сохраняя при этом операции затенения отдельно от операций геометрии. Она необязательно может завершить луч с помощью встроенной функции rtTerminateRay, которая будет останавливать все обходы и освободить стек вызовов до самого последнего вызова rtTrace. Это легковесный механизм исключение, который может быть использован для реализации раннего обрыва теневых лучей и пространственной непроходимости. Кроме того, программа любого попадания может игнорировать пересечение используя rtIgnoreIntersection, которая позволяет обходу продолжать искать другие геометрические объекты. Пересечение может быть проигнорировано, например, на основе текстуры поиска канала, тем самым реализация эффективной альфа-прозрачности отображается без перезагрузки обхода.

```
RT_PROGRAM void pinhole_camera() {
Ray ray = PinholeCamera::makeRay( launchIndex );
UserPayload payload;
rtTrace( topObject, ray, payload );
outputBuffer[launchIndex] = payload.result;
}
```

Рисунок 5 – Пример программы генерации луча (в CUDA C) для одного сэмпла на пиксель. Расположение 2-мерной сетки вызова программы дает семантическую переменную launchIndex, которая используется для создания первичного луча с помощью модели pinhole камеры. Во время трассировки луча, вызванные программы попадания материала заполняют результирующее поле в определяемой пользователем структуре payload. Переменная topObject указывает на месторасположение в иерархии сцены, где луч обхода должен начинаться, как правило, в корне узла графа. На месте, указанном launchIndex, результат записывается в выходной буфер, который будет отображаться в приложении.

Программы промаха выполняются, когда луч не пересекает любую геометрию в предоставленном интервале. Они могут быть использованы для реализации цвета фона или подстановки карты среды .

Исключение программы выполняются, когда система обнаруживает состояние исключения, например, когда стек рекурсии превышает объем памяти, доступный для каждого потока, или когда индекс доступа к буферу вне диапазона. OptiX также поддерживает определяемые пользователем исключения, которые могут быть сгенерированы из любой программы. Программа исключение может реагировать, например, с помощью печати диагностических сообщений или визуализации состояний путем записи специальных цветовых значений в буфер вывода пикселя. Программы выбора обхода подвергаются программированию для низкого уровня узла обхода графа. Например, приложение может выбрать изменение уровня геометрической детализации для частей сцены на Perray основе. В этом случае, программа выбора будет рассматривать расстояние лучей или дифференциал лучей, который хранится с полезной нагрузкой и принимает решение обхода на основе этих данных.

2.3.2 Представление сцены

Движок OptiX использует гибкую структуру для представления информации сцены и связанные программируемые операций, собранные в объекте контейнера называнные контекстом. Это представление тоже является механизмом для привязки программируемых шейдеров к данным конкретного объекта, которые требуются.

Узлы иерархии. Сцена представлена как график. Это представление очень легковесное и управляет обходом лучей через сцену. Оно также может быть использовано для реализации двухуровневой иерархии сущностей для анимации твердых предметов или других общих структур сцены. Для поддержки создания экземпляров и обмена общими данными, узлы могут иметь несколько родителей. Четыре основных типов узлов могут быть использованы, чтобы обеспечить представление сцены с помощью ориентированного графа. Любой узел может быть использован в качестве корневого узла обхода. Это позволяет, например, создавать различные представления, которые будут использоваться для различных типов лучей.

Группа узлов содержать ноль или более (но обычно два или больше) потомков любого типа узла. Узел группы имеет структуру ускорения, связанный с ним, и может быть использовано, чтобы обеспечить верхний уровень структуры двухуровневого обхода. Геометрическая группа узлов является потомками на графике и содержат примитивные и материальные объекты, описанные ниже. Этот тип узла также имеет структуру ускорения связанную с ним. Любая непустая сцена будет содержать хотя бы одну геометрическую группу. Преобразованные узлы имеют один дочерний элемент любого типа узла, а также связанную матрицу 4×3, которая используется для выполнения преобразованиz базовой геометрии. Выбранные узлы имеют ноль или более потомков любого типа узла, плюс одну программу обхода, которая выполняется для выбора среди имеющихся потомков. Хотя это и не реализовано в текущей версии Optix библиотек, узел граф может быть циклическим, если узел селектора использовать осторожно, чтобы избежать бесконечной рекурсии.

Геометрические и материальные объекты. Основная часть данных хранится в геометрии узлов в потомках графа. Они содержат объекты, которые определяют геометрию и затенение операции. Они могут также иметь несколько родителей, позволяя материалам и геометрии осуществлять обмен информацией в нескольких точках на графе.

Объекты Геометрической сущности связан с геометрическим объектом набором материальных объектов. Это общая структура, используемая графами сцены позволяет сохранить геометрическую и теневую информацию ортогонально.

Объекты материала храненят информацию о операциях затенения, включая программы, вызываемые для каждого пересечения по мере их обнаружения (программа любого попадания) и для ближайшего пересечения к началу данного луча (программа ближайшего попадания).

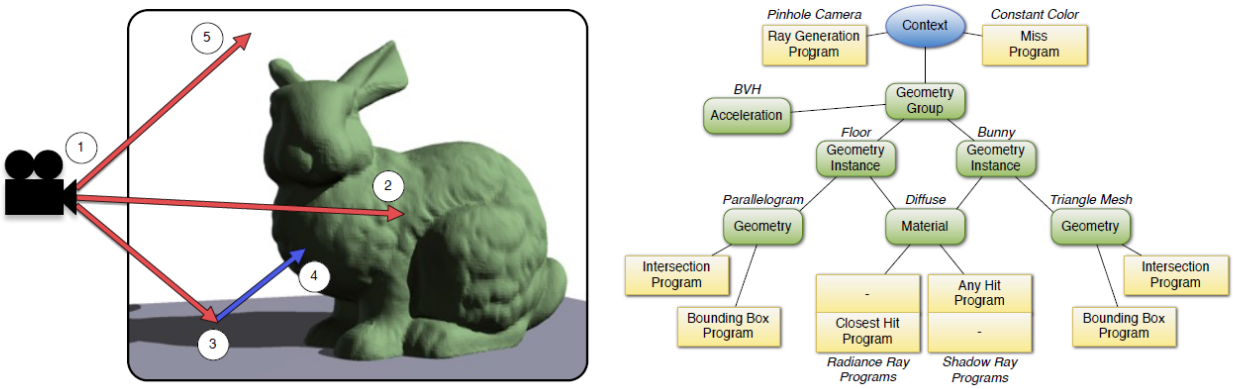

Рисунок 6 – Справа: Полный контекст OptiX простой сцены с помощью pinhole камеры, двух объектов и теней. Программа генерации луча реализует камеру, в то время как программа промаха реализует постоянный белый фон. Геометрия одной группы содержит два сущности геометрии с одной BVH построеной через всю основную геометрию в треугольной сетке и плоскостях. Два типа геометрии будут реализованы, треугольная сетка и параллелограмм, каждый со своим набором пересечения и программе ограничивающей рамки. Два экземпляра геометрии совместно используют один материал, который реализует диффузную модель освещения и полностью ослабляет теневые лучи через программы ближайшего попадания и любого попадания соответственно. Слева: Выполнение программ. 1. Программа генерации луча создает лучи и проводит их через группы геометрии, который инициирует BVH обход. 2. Если луч пересекается с геометрией,программа ближайшего попадания будет вызываться после того как точка попадания будет найдена. 3. Материал будет порождать теневые лучи и простраивать их от геометрии сцены. 4. Когда пересечение вдоль теневого луча найдено, программа любого попадания прекращает обход луча и возвращает в вызывающую программу с информацией о тени. 5. Если луч не пересекает с какой-либо геометрией сцены, будет вызываться программа промаха.

2.3.3 Объектная модель и модель данных

OptiX использует объектную модель специального назначения, предназначенную для минимизации постоянных данных, используемых в программных операциях. В отличие от системы OpenGL, где только одна комбинация шейдеров используется одновременно. Тем не менее, трассировка лучей может случайно получить доступ к данным материала и объекта. Поэтому вместо общих переменных, используемых в шейдерных языках OpenGL, OptiX позволяет описать любой объект и узел, описанный выше задав произвольный набор переменных, выраженных как пара ключ-значение. Переменные устанавливаются клиентским приложением и имеют доступ только для чтения во время выполнения трассировки. Переменные могут быть скалярного или векторного целого типа или с плавающей точкой (например, float3, int4), а также определяемые пользователем Структуры и ссылки на буферы и текстурные образцы.

Механизм наследования для этих переменных является уникальным для OptiX. Вместо ос-

новных классов модели наследования с одной сущностью или указателем на него, движок OptiX отслеживает текущую геометрию и материальные объекты и текущий узел обхода. Значения переменных наследуются от объектов, которые являются активными в каждой точке потока управления. Например, программа пересечения унаследует определения от геометрии и геометрических объектов, в дополнение к глобальным переменным, определенным в контексте. Концептуально, OptiX рассматривает каждый из этих объектов как пару ключ / значение, когда есть доступ к переменным. Этот механизм, который можно рассматривать как обобщение вложенной области видимости, можно найти в большинстве языков программирования. Он также может быть реализован достаточно эффективно в JIT компиляторе.

В качестве примера, как это используется, рассмотрим массив источников света, называемых светом. Как правило, пользователь будет определять свет в контексте, общий OptiX. Это присваивает данное значение для всех шейдеров в всех сценах. Однако, если свет для конкретного объекта необходимо переопределить, другая переменная с этим же именем может быть создана и присоединена к экземпляру геометрии, связанной с этим объектом. Таким образом, программы, связанные с этим объектом будет использовать переопределенное значение света, а не значения, прикрепленное к контексту. Это мощный механизм, который может быть использован для минимизации данных сцены, что обеспечивают высокую производительность на архитектурах с минимальными кэшем. Манера, в которой обрабатываются эти случаи может значительно варьироваться от одной визуализации к другой, так что движок OptiX обеспечивает базовую функциональность, чтобы выразить любое количество правил переопределения эффективно.

Особый тип переменной, отмеченного атрибутом ключевых слов можно использовать для передачи информации из программы пересечения с программой ближайшего и любого попадания. Они аналогичны varying variables в OpenGL, и используются для передачи текстурных координат, нормалей и другой информации о затенении от программ пересечения к программами затенения. Эти переменные имеют специальную семантику-они написаны программой пересечения, но только хранятся значения, связанные с ближайшей точкой пересечения. Этот механизм позволяет операциям пересечения быть полностью отделеными от операций затенения, включая несколько подобных примитивов и / или сетки форматов хранения в то же время поддерживая текстурирование, затенение нормалейи и объектов кривизны для разносных лучей. Атрибуты, которые не используются любой программой ближайшего или любого-попадания могут быть опущены компилятором OptiX.

2.3.4 Динамическая отправка

Чтобы позволить нескольким операциям трассировки лучей сосуществовать в одном исполнении, OptiX использует пользовательский тип луча. Тип луча это просто индекс, который выбирает определенный набор слотов для программ любого и ближайшего попадания, который будет выполнен, если точка пересечения найдена. Это может быть использовано, например, для обработки теневых лучей отдельно от других лучей.

Точно так же, несколько точек входа в контексте OptiX включают эффективный способ

представлять различные проходы по тем же набором геометрии. Например, маппер фотона может использовать одну точку входа, чтобы излучить фотоны в сцену и вторую точку входа для просмотра излученных лучей.

2.3.5 Буферы и текстуры

Ключевая абстракция для хранения больших объемов данных является объектом многомерного буфера, который представляет собой 1 -, 2 - или 3-мерный массив фиксированного размера элемента. Буфер можно получить через объект C+ + обертки в любой из программ. Буферы могут быть доступны только для чтения, только для записи или для чтения и записи и поддерживать атомарные операции, когда поддерживаются аппаратно. Буфер основанный на дескрипторе не предоставляет необработанные указатели, что позволяет среде выполнения OptiX в реальном времени переразмещать буферы для хранения уплотнения, или для продвижения к другим пространствам памяти для производительности. Буферы обычно используются для вывода изображений, данных треугольников, списков источников света и других данных на основе массивов. Буферы являются единственным средством вывода данных из программы OptiX. В большинстве случаев, программа генерации луча будет отвечать за запись данных в выходной буфер, но любой из программ Optix разрешается писать в выходные буферы в любом месте, но без каких-либо гарантий очередности. Буфер также может быть привязан к объекту текстурного сэмплера, который будет использовать аппаратное GPU текстурирование. Буферы и объекты текстурного сэмплера обязаны быть Optix переменными и использовать те же механизмы сбора данных как шейдерных значений. Кроме того, оба буфера и блока выборки могут взаимодействовать с OpenGL и DirectX, включая эффективную реализацию гибридных растеризации / трассировки лучей приложения.

3 Практическая часть

3.1 Примеры

что то написать.

3.2 Выполнение работы

```
#include <optixu/optixpp_namespace.h>
#include <optixu/optixu_math_namespace.h>
#include <optixu/optixu_aabb_namespace.h>
#include <sutil.h>
#include <GLUTDisplay.h>
#include <PlyLoader.h>
#include <ObjLoader.h>
#include "commonStructs.h"
#include <string>
```

Рисунок 7 – пример 1.

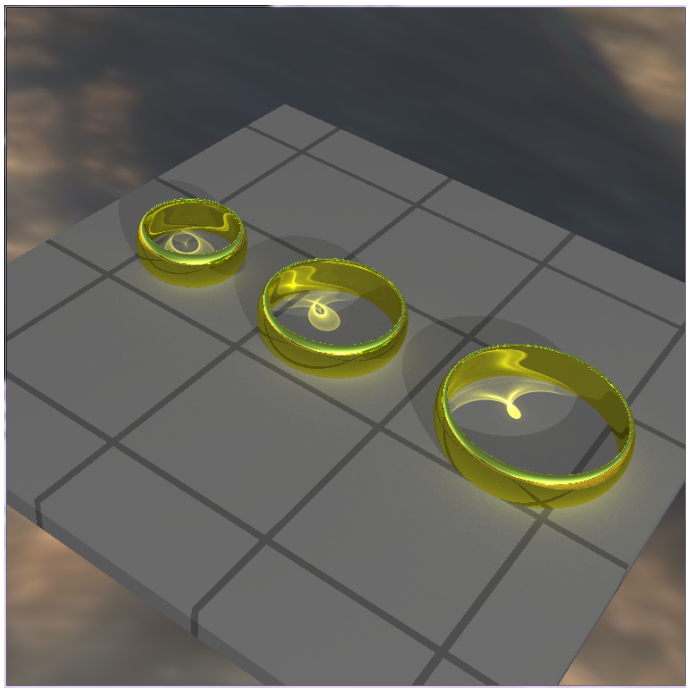

Рисунок 8 – Пример 2.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cstring>
#include "random.h"
#include "MeshScene.h"

using namespace optix;

class MeshViewer : public MeshScene
{
```

```
public:
  //
  // Helper types
  //
  enum ShadeMode
  {
    SM_PHONG=0,
    SM_AO,
    SM_NORMAL,
    SM_ONE_BOUNCE_DIFFUSE,
    SM_AO_PHONG
  };

  enum CameraMode
  {
    CM_PINHOLE=0,
    CM_ORTHO
  };

  //
  // MeshViewer specific
  //
  MeshViewer();

  // Setters for controlling application behavior
  void setShadeMode( ShadeMode mode ) { m_shade_mode = mode;}
  void setCameraMode( CameraMode mode ) { m_camera_mode = mode;}
  void setAORadius( float ao_radius ) { m_ao_radius = ao_radius;}
  void setAOSampleMultiplier( int ao_sample_mult ) { m_ao_sample_mult = ao_sample_mult; }
  void setLightScale( float light_scale ) { m_light_scale = light_scale;}
  void setAA( bool onoff ) { m_aa_enabled = onoff;}
  void setAnimation( bool anim ) { m_animation = anim;}

  virtual void   initScene( InitialCameraData& camera_data );
  virtual void   doResize( unsigned int width, unsigned int height );
  virtual void   trace( const RayGenCameraData& camera_data );
  virtual void   cleanUp();
  virtual bool   keyPressed(unsigned char key, int x, int y);
  virtual Buffer getOutputBuffer();

private:
```

```
  void initContext();
  void initLights();
  void initMaterial();
  void initGeometry();
  void initCamera( InitialCameraData& cam_data );
  void preprocess();
  void resetAccumulation();
  void genRndSeeds( unsigned int width, unsigned int height );

  CameraMode    m_camera_mode;

  ShadeMode  m_shade_mode;
  bool    m_aa_enabled;
  float    m_ao_radius;
  int       m_ao_sample_mult;
  float    m_light_scale;

  Material  m_material;
  Aabb      m_aabb;
  Buffer     m_rnd_seeds;
  Buffer     m_accum_buffer;
  bool       m_accum_enabled;

  float       m_scene_epsilon;
  int          m_frame;
  bool        m_animation;
};
MeshViewer::MeshViewer():
  MeshScene ( false, false, false ),
  m_camera_mode ( CM_PINHOLE ),
  m_shade_mode ( SM_PHONG ),
  m_aa_enabled ( false ),
  m_ao_radius ( 1.0f ),
  m_ao_sample_mult ( 1 ),
  m_light_scale ( 1.0f ),
  m_accum_enabled ( false ),
  m_scene_epsilon ( 1e-4f ),
  m_frame ( 0 ),
  m_animation ( false )
{
}
```

```
void MeshViewer::initScene( InitialCameraData& camera_data )
{
  initContext();
  initLights();
  initMaterial();
  initGeometry();
  initCamera( camera_data );
  preprocess();
}
void MeshViewer::initContext()
{
  m_context->setRayTypeCount( 3 );
  m_context->setEntryPointCount( 1 );
  m_context->setStackSize( 1180 );

  m_context[ "radiance_ray_type" ]->setUint( 0u );
  m_context[ "shadow_ray_type" ]->setUint( 1u );
  m_context[ "max_depth" ]->setInt( 5 );
  m_context[ "ambient_light_color" ]->setFloat( 0.2f, 0.2f, 0.2f );
  m_context[ "output_buffer" ]->set( createOutputBuffer(RT_FORMAT_UNSIGNED_BYTE4, WIDTH,
  m_context[ "jitter_factor" ]->setFloat( m_aa_enabled ? 1.0f : 0.0f );

  m_accum_enabled = m_aa_enabled ||
                    m_shade_mode == SM_AO ||
                    m_shade_mode == SM_ONE_BOUNCE_DIFFUSE ||
                    m_shade_mode == SM_AO_PHONG;

  // Ray generation program setup
  const std::string camera_name = m_camera_mode == CM_PINHOLE ? "pinhole_camera" : "ortho
  const std::string camera_file = m_accum_enabled? "accum_camera.cu" :
                                  m_camera_mode == CM_PINHOLE ? "pinhole_camera.cu"  :
                                                                "orthographic_camera.cu";

  if( m_accum_enabled ) {
    // The raygen program needs accum_buffer
    m_accum_buffer = m_context->createBuffer( RT_BUFFER_INPUT_OUTPUT | RT_BUFFER_GPU_LOCA
                                              WIDTH, HEIGHT );
    m_context["accum_buffer"]->set( m_accum_buffer );
    resetAccumulation();
  }
```

```
  const std::string camera_ptx  = ptxpath( "sample6", camera_file );
  Program ray_gen_program = m_context->createProgramFromPTXFile( camera_ptx, camera_name
  m_context->setRayGenerationProgram( 0, ray_gen_program );


  // Exception program
  const std::string except_ptx  = ptxpath( "sample6", camera_file );
  m_context->setExceptionProgram( 0, m_context->createProgramFromPTXFile( except_ptx, "ex
  m_context[ "bad_color" ]->setFloat( 0.0f, 1.0f, 0.0f );


  // Miss program
  const std::string miss_ptx = ptxpath( "sample6", "constantbg.cu" );
  m_context->setMissProgram( 0, m_context->createProgramFromPTXFile( miss_ptx, "miss" ) )
  m_context[ "bg_color" ]->setFloat(  0.34f, 0.55f, 0.85f );
}


void MeshViewer::initLights()
{
  // Lights buffer
  BasicLight lights[] = {
    { make_float3( -60.0f,  30.0f, -120.0f ), make_float3( 0.2f, 0.2f, 0.25f )*m_light_sc
    { make_float3( -60.0f,   0.0f,  120.0f ), make_float3( 0.1f, 0.1f, 0.10f )*m_light_sc
    { make_float3(  60.0f,  60.0f,   60.0f ), make_float3( 0.7f, 0.7f, 0.65f )*m_light_sc
  };

  Buffer light_buffer = m_context->createBuffer(RT_BUFFER_INPUT);
  light_buffer->setFormat(RT_FORMAT_USER);
  light_buffer->setElementSize(sizeof( BasicLight ) );
  light_buffer->setSize( sizeof(lights)/sizeof(lights[0]) );
  memcpy(light_buffer->map(), lights, sizeof(lights));
  light_buffer->unmap();

  m_context[ "lights" ]->set( light_buffer );
}


void MeshViewer::initMaterial()
{
  switch( m_shade_mode ) {
```

```
case SM_PHONG: {
  // Use the default obj_material created by ObjLoader
  break;
}

case SM_NORMAL: {
  const std::string ptx_path = ptxpath("sample6", "normal_shader.cu");
  m_material = m_context->createMaterial();
  m_material->setClosestHitProgram( 0, m_context->createProgramFromPTXFile( ptx_path
  break;
}

case SM_AO: {
  const std::string ptx_path = ptxpath("sample6", "ambocc.cu");
  m_material = m_context->createMaterial();
  m_material->setClosestHitProgram( 0, m_context->createProgramFromPTXFile( ptx_path
  m_material->setAnyHitProgram    ( 1, m_context->createProgramFromPTXFile( ptx_path
  break;
}

case SM_ONE_BOUNCE_DIFFUSE: {
  const std::string ptx_path = ptxpath("sample6", "one_bounce_diffuse.cu");
  m_material = m_context->createMaterial();
  m_material->setClosestHitProgram( 0, m_context->createProgramFromPTXFile( ptx_path
  m_material->setAnyHitProgram    ( 1, m_context->createProgramFromPTXFile( ptx_path
  break;
}

case SM_AO_PHONG: {
  const std::string ptx_path = ptxpath("sample6", "ambocc.cu");
  m_material = m_context->createMaterial();
  m_material->setClosestHitProgram( 0, m_context->createProgramFromPTXFile( ptx_path
  m_material->setAnyHitProgram    ( 1, m_context->createProgramFromPTXFile( ptx_path
  m_material->setAnyHitProgram    ( 2, m_context->createProgramFromPTXFile( ptx_path
  m_context["Kd"]->setFloat(1.0f);
  m_context["Ka"]->setFloat(0.6f);
  m_context["Ks"]->setFloat(0.0f);
  m_context["Kr"]->setFloat(0.0f);
  m_context["phong_exp"]->setFloat(0.0f);
  break;
}
```

```
  }

  if( m_accum_enabled ) {
    genRndSeeds( WIDTH, HEIGHT );
  }
}


void MeshViewer::initGeometry()
{
  double start, end;
  sutilCurrentTime(&start);

  m_geometry_group = m_context->createGeometryGroup();

  if( ObjLoader::isMyFile( m_filename.c_str() ) ) {
    // Load OBJ model
    ObjLoader* loader = 0;
    if( m_shade_mode == SM_NORMAL || m_shade_mode == SM_AO || m_shade_mode == SM_AO_PHONG
      loader = new ObjLoader( m_filename.c_str(), m_context, m_geometry_group, m_material
    } else if ( m_shade_mode == SM_ONE_BOUNCE_DIFFUSE ) {
      loader = new ObjLoader( m_filename.c_str(), m_context, m_geometry_group, m_material
    } else {
      loader = new ObjLoader( m_filename.c_str(), m_context, m_geometry_group, m_accel_bu
    }
    loader->load();
    m_aabb = loader->getSceneBBox();
    delete loader;

  } else if( PlyLoader::isMyFile( m_filename ) ) {
    // Load PLY model
    PlyLoader loader( m_filename, m_context, m_geometry_group, m_material, m_accel_builde
    loader.load();
    m_aabb = loader.getSceneBBox();

  } else {
    std::cerr << "Unrecognized model file extension '" << m_filename << "'" << std::endl;
    exit( 0 );
  }

  // Load acceleration structure from a file if that was enabled on the
```

```
  // command line, and if we can find a cache file. Note that the type of
  // acceleration used will be overridden by what is found in the file.
  loadAccelCache();


  m_context[ "top_object" ]->set( m_geometry_group );
  m_context[ "top_shadower" ]->set( m_geometry_group );

  sutilCurrentTime(&end);
  std::cerr << "Time to load " << (m_accel_large_mesh ? "and cluster " : "") << "geometry
}


void MeshViewer::initCamera( InitialCameraData& camera_data )
{
  // Set up camera
  float max_dim  = m_aabb.maxExtent();
  float3 eye     = m_aabb.center();
  eye.z         += 2.0f * max_dim;

  camera_data = InitialCameraData( eye,  // eye
                                   m_aabb.center(), // lookat
                                   make_float3( 0.0f, 1.0f, 0.0f ), // up
                                   30.0f );                         // vfov

  // Declare camera variables.  The values do not matter, they will be overwritten in tra
  m_context[ "eye"]->setFloat( make_float3( 0.0f, 0.0f, 0.0f ) );
  m_context[ "U"  ]->setFloat( make_float3( 0.0f, 0.0f, 0.0f ) );
  m_context[ "V"  ]->setFloat( make_float3( 0.0f, 0.0f, 0.0f ) );
  m_context[ "W"  ]->setFloat( make_float3( 0.0f, 0.0f, 0.0f ) );

}


void MeshViewer::preprocess()
{
  // Settings which rely on previous initialization
  m_scene_epsilon = 1.e-4f * m_aabb.maxExtent();
  m_context[ "scene_epsilon"      ]->setFloat( m_scene_epsilon );
  m_context[ "occlusion_distance" ]->setFloat( m_aabb.maxExtent() * 0.3f * m_ao_radius );
```

```
  // Prepare to run
  m_context->validate();
  double start, end_compile, end_AS_build;
  sutilCurrentTime(&start);
  m_context->compile();
  sutilCurrentTime(&end_compile);
  std::cerr << "Time to compile kernel: "<<end_compile-start<<" s.\n";
  m_context->launch(0,0);
  sutilCurrentTime(&end_AS_build);
  std::cerr << "Time to build AS      : "<<end_AS_build-end_compile<<" s.\n";

  // Save cache file
  saveAccelCache();
}


bool MeshViewer::keyPressed(unsigned char key, int x, int y)
{
   switch (key)
   {
     case 'e':
       m_scene_epsilon *= .1f;
       std::cerr << "scene_epsilon: " << m_scene_epsilon << std::endl;
       m_context[ "scene_epsilon" ]->setFloat( m_scene_epsilon );
       return true;
     case 'E':
       m_scene_epsilon *= 10.0f;
       std::cerr << "scene_epsilon: " << m_scene_epsilon << std::endl;
       m_context[ "scene_epsilon" ]->setFloat( m_scene_epsilon );
       return true;
   }
   return false;
}


void MeshViewer::doResize( unsigned int width, unsigned int height )
{
  // output_buffer resizing handled in base class
  if( m_accum_enabled ) {
    m_accum_buffer->setSize( width, height );
    m_rnd_seeds->setSize( width, height );
```

```
    genRndSeeds( width, height );
    resetAccumulation();
  }
}

void MeshViewer::trace( const RayGenCameraData& camera_data )
{
  if (m_animation && GLUTDisplay::isBenchmark() ) {
    static float angleU = 0.0f, angleV = 0.0f, scale = 1.0f, dscale = 0.96f, backside = 0
    static int phase = 0, accumed_frames = 0;
    const float maxang = M_PIf * 0.2f;
    const float rotvel = M_2_PIf*0.1f;
    float3 c = m_aabb.center();
    float3 e = camera_data.eye;

    Matrix3x3 m = make_matrix3x3(Matrix4x4::rotate(angleV + backside, normalize(camera_da
      Matrix4x4::rotate(angleU, normalize(camera_data.U)) * Matrix4x4::scale(make_float3(

    if( !m_accum_enabled || accumed_frames++ > 5 ) { // Accumulate 5 frames per animation
      switch(phase) {
      case 0: angleV += rotvel; if(angleV > maxang) { angleV =  maxang; phase++; } break;
      case 1: angleU += rotvel; if(angleU > maxang) { angleU =  maxang; phase++; } break;
      case 2: angleV -= rotvel; if(angleV <-maxang) { angleV = -maxang; phase++; } break;
      case 3: angleU -= rotvel; if(angleU <-maxang) { angleU = -maxang; phase=0; } break;
      }

      scale *= dscale;
      if(scale < 0.1f) { dscale = 1.0f / dscale; backside = M_PIf - backside; }
      if(scale > 1.0f) { dscale = 1.0f / dscale; }

      accumed_frames = 0;
      m_camera_changed = true;
    }

    m_context["eye"]->setFloat( c-m*(c-e) );
    m_context["U"]->setFloat( m*camera_data.U );
    m_context["V"]->setFloat( m*camera_data.V );
    m_context["W"]->setFloat( m*camera_data.W );
  } else {
    m_context["eye"]->setFloat( camera_data.eye );
    m_context["U"]->setFloat( camera_data.U );
```

```
    m_context["V"]->setFloat( camera_data.V );
    m_context["W"]->setFloat( camera_data.W );
  }

  Buffer buffer = m_context["output_buffer"]->getBuffer();
  RTsize buffer_width, buffer_height;
  buffer->getSize( buffer_width, buffer_height );

  if( m_accum_enabled && !m_camera_changed ) {
    // Use more AO samples if the camera is not moving, for increased !/$.
    // Do this above launch to avoid overweighting the first frame
    m_context["sqrt_occlusion_samples"]->setInt( 3 * m_ao_sample_mult );
    m_context["sqrt_diffuse_samples"]->setInt( 3 );
  }

  m_context->launch( 0, static_cast<unsigned int>(buffer_width), static_cast<unsigned int

  if( m_accum_enabled ) {
    // Update frame number for accumulation.
    ++m_frame;
    if( m_camera_changed ) {
      m_camera_changed = false;
      resetAccumulation();
    }

    // The frame number is used as part of the random seed.
    m_context["frame"]->setInt( m_frame );
  }
}
void MeshViewer::cleanUp()
{
  SampleScene::cleanUp();
}
Buffer MeshViewer::getOutputBuffer()
{
  return m_context["output_buffer"]->getBuffer();
}


void MeshViewer::resetAccumulation()
{
```

```
  m_frame = 0;
  m_context[ "frame"                   ]->setInt( m_frame );
  m_context[ "sqrt_occlusion_samples" ]->setInt( 1 * m_ao_sample_mult );
  m_context[ "sqrt_diffuse_samples"   ]->setInt( 1 );
}


void MeshViewer::genRndSeeds( unsigned int width, unsigned int height )
{
  // Init random number buffer if necessary.
  if( m_rnd_seeds.get() == 0 ) {
    m_rnd_seeds = m_context->createBuffer( RT_BUFFER_INPUT_OUTPUT | RT_BUFFER_GPU_LOCAL,
                                           WIDTH, HEIGHT);
    m_context["rnd_seeds"]->setBuffer(m_rnd_seeds);
  }

  unsigned int* seeds = static_cast<unsigned int*>( m_rnd_seeds->map() );
  fillRandBuffer(seeds, width*height);
  m_rnd_seeds->unmap();
}

void printUsageAndExit( const std::string& argv0, bool doExit = true )
{
  std::cerr
    << "Usage  : " << argv0 << " [options]\n"
    << "App options:\n"
    << "  -h  | --help                                Print this usage message\n"
    << "  -o  | --obj <obj_file>                      Specify .OBJ model to be rendered\n"
    << "  -c  | --cache                               Turn on acceleration structure cach
    << "  -a  | --ao-shade                            Use progressive ambient occlusion sh
    << "  -ap | --ao-phong-shade                      Use progressive ambient occlusion ar
    << "  -aa | --antialias                           Use subpixel jittering to perform ar
    << "  -n  | --normal-shade                        Use normal shader\n"
    << "  -i  | --diffuse-shade                       Use one bounce diffuse shader\n"
    << "  -O  | --ortho                               Use orthographic camera (cannot use
    << "  -r  | --ao-radius <scale>                   Scale ambient occlusion radius\n"
    << "  -m  | --ao-sample-mult <n>                  Multiplier for the number of AO samp
    << "  -l  | --light-scale <scale>                 Scale lights by constant factor\n"
    << "        --large-mesh                          Massive dataset mode\n"
    << "        --animation                           Spin the model (useful for benchmarl
    << "        --trav <name>                         Acceleration structure traverser\n"
```

```
    << "        --build <name>                        Acceleration structure builder\n"
    << "        --refine <n>                          Acceleration structure refinement pa
    << std::endl;
  GLUTDisplay::printUsage();

  std::cerr
    << "App keystrokes:\n"
    << "  e Decrease scene epsilon size (used for shadow ray offset)\n"
    << "  E Increase scene epsilon size (used for shadow ray offset)\n"
    << std::endl;

  if ( doExit ) exit(1);
}


int main( int argc, char** argv )
{
  GLUTDisplay::init( argc, argv );

  GLUTDisplay::contDraw_E draw_mode = GLUTDisplay::CDNone;
  MeshViewer scene;
  scene.setMesh( (std::string( sutilSamplesDir() ) + "/simpleAnimation/cow.obj").c_str()

  for ( int i = 1; i < argc; ++i ) {
    std::string arg( argv[i] );
    if ( arg == "-c" || arg == "--cache" ) {
      scene.setAccelCaching( true );
    } else if( arg == "-n" || arg == "--normal-shade" ) {
      scene.setShadeMode( MeshViewer::SM_NORMAL );
    } else if( arg == "-a" || arg == "--ao-shade" ) {
      scene.setShadeMode( MeshViewer::SM_AO);
      draw_mode = GLUTDisplay::CDProgressive;
    } else if( arg == "-i" || arg == "--diffuse-shade" ) {
      scene.setShadeMode( MeshViewer::SM_ONE_BOUNCE_DIFFUSE );
      draw_mode = GLUTDisplay::CDProgressive;
    } else if( arg == "-ap" || arg == "--ao-phong-shade" ) {
      scene.setShadeMode( MeshViewer::SM_AO_PHONG );
      draw_mode = GLUTDisplay::CDProgressive;
    } else if( arg == "-aa" || arg == "--antialias" ) {
      scene.setAA( true );
      draw_mode = GLUTDisplay::CDProgressive;
```

```
} else if( arg == "-O" || arg == "--ortho" ) {
  scene.setCameraMode( MeshViewer::CM_ORTHO );
} else if( arg == "-h" || arg == "--help" ) {
  printUsageAndExit( argv[0] );
} else if( arg == "-o" || arg == "--obj" ) {
  if ( i == argc-1 ) {
    printUsageAndExit( argv[0] );
  }
  scene.setMesh( argv[++i] );
} else if( arg == "--trav" ) {
  if ( i == argc-1 ) {
    printUsageAndExit( argv[0] );
  }
  scene.setTraverser( argv[++i] );
} else if( arg == "--build" ) {
  if ( i == argc-1 ) {
    printUsageAndExit( argv[0] );
  }
  scene.setBuilder( argv[++i] );
} else if( arg == "--refine" ) { // N tree rotation passes to improve BVH quality
  if ( i == argc-1 ) {
    printUsageAndExit( argv[0] );
  }
  scene.setRefine( argv[++i] );
} else if( arg == "--kd" ) {     // Keep this arg for a while for backward compatibil
  scene.setBuilder( "TriangleKdTree" );
  scene.setTraverser( "KdTree" );
} else if( arg == "--lbvh" ) {   // Keep this arg for a while for backward compatibil
  scene.setBuilder( "Lbvh" );
} else if( arg == "--bvh" ) {    // Keep this arg for a while for backward compatibil
  scene.setBuilder( "Bvh" );
} else if( arg == "--large-mesh" ) {
  scene.setLargeMesh( true );
} else if( arg == "--animation" ) {
  scene.setAnimation( true );
} else if( arg == "-r" || arg == "--ao-radius" ) {
  if ( i == argc-1 ) {
    printUsageAndExit( argv[0] );
  }
  scene.setAORadius( static_cast<float>( atof( argv[++i] ) ) );
} else if( arg == "-m" || arg == "--ao-sample-mult" ) {
```

```
      if ( i == argc-1 ) {
        printUsageAndExit( argv[0] );
      }
      scene.setAOSampleMultiplier( atoi( argv[++i] ) );
    } else if( arg == "-l" || arg == "--light-scale" ) {
      if ( i == argc-1 ) {
        printUsageAndExit( argv[0] );
      }
      scene.setLightScale( static_cast<float>( atof( argv[++i] ) ) );
    } else {
      std::cerr << "Unknown option: '" << arg << "'" << std::endl;
      printUsageAndExit( argv[0] );
    }
  }

  if( !GLUTDisplay::isBenchmark() ) printUsageAndExit( argv[0], false );

  try {
    GLUTDisplay::run( "OptiX Viewer", &scene, draw_mode );
  } catch( Exception& e ){
    sutilReportError( e.getErrorString().c_str() );
    exit(1);
  }

  return 0;
}
```

Результат работы:

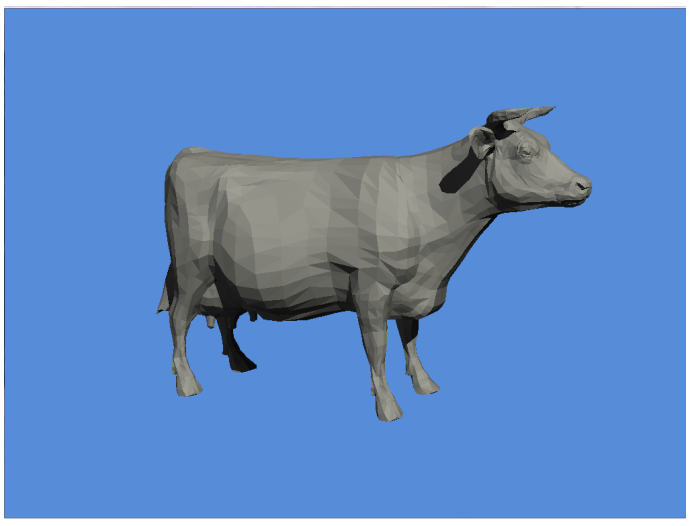

Рисунок 9 – Пример 2.

4 Заключение

OptiX — проблемно-ориентированный своевременный компилятор, который генерирует пользовательские ядра трассировки лучей, комбинируя предоставленные пользователями программы для генерации луча, заливки материала, объектного пересечения и обхода сцены. Механизм OptiX основывается на базовом наблюдении, что большинство алгоритмов трассировки лучей могут быть реализованы, используя маленький набор программируемых операций. OptiX достигает высокой производительности через компактную объектную модель и применения нескольких специфичной для трассировки лучей оптимизаций компилятора. Для простоты использования OptiX представляет модель программирования единственного луча с полной поддержкой рекурсии и динамического механизма отправки, подобного вызовам виртуальной функции. Механизм OptiX демонстрирует, что большинство алгоритмов трассировки лучей может быть реализовано, используя маленький набор легких программируемых операций. Механизм OptiX обеспечивает такие механизмы выполнения, что программисты пользуются знакомыми методами работы с трассировкой лучами и не обременяют себя низкоуровневой оптимизацией. Механизм OptiX реализует объектную модель, которая использует динамическое наследование, чтобы упростить компактное представление параметров сцены. Система OptiX обеспечивает и высокоэффективный API трассировки лучей общего назначения. OptiX балансирует простоту использования с производительности, представляя простую модель программирования, на основе программируемого конвейера трассировки лучей для пользовательских программ единственного луча, которые могут быть скомпилированы в эффективное мегаядро самопланирования.Таким образом основа OptiX - JIT-компилятор, который обрабатывает программы, отрывки определенного пользователями кода на языке PTX.

Список литературы

1. Д. Чеканов. Метод трассировки лучей против растеризации: новое поколение качества графики? — Информационный портал Tom's Hardware

2. Боресков А. В. Основы CUDA — Информационный портал steps3d.narod.ru

3. Биглер Дж., Стивенс, А., и Паркер, С. Г. 2006. Разработка систем с параллельной интерактивной трассировки лучей. — Симпозиум по интерактивной трассировки лучей, 187-196 2006.

4. Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, Martin Stich. OptiX: A General Purpose Ray Tracing Engine, ACM Transactions on Graphics, 2010

5. NVIDIA 2010. Программирование NVIDIA OptiX движка трассировки лучей Руководство Версия 2.0. (http://developer.nvidia.com/object/-OptiX-home.html.)