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. [1]

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).

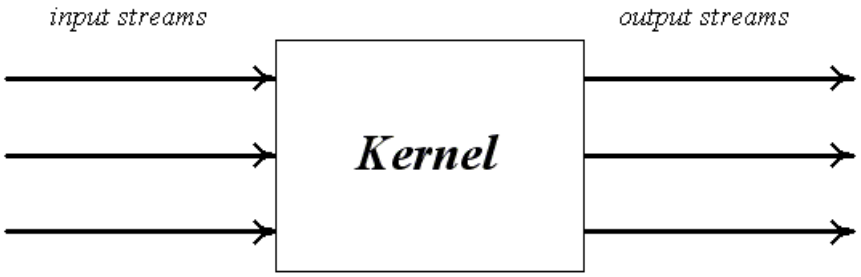

Рисунок 1 – Потоковые вычисления

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)

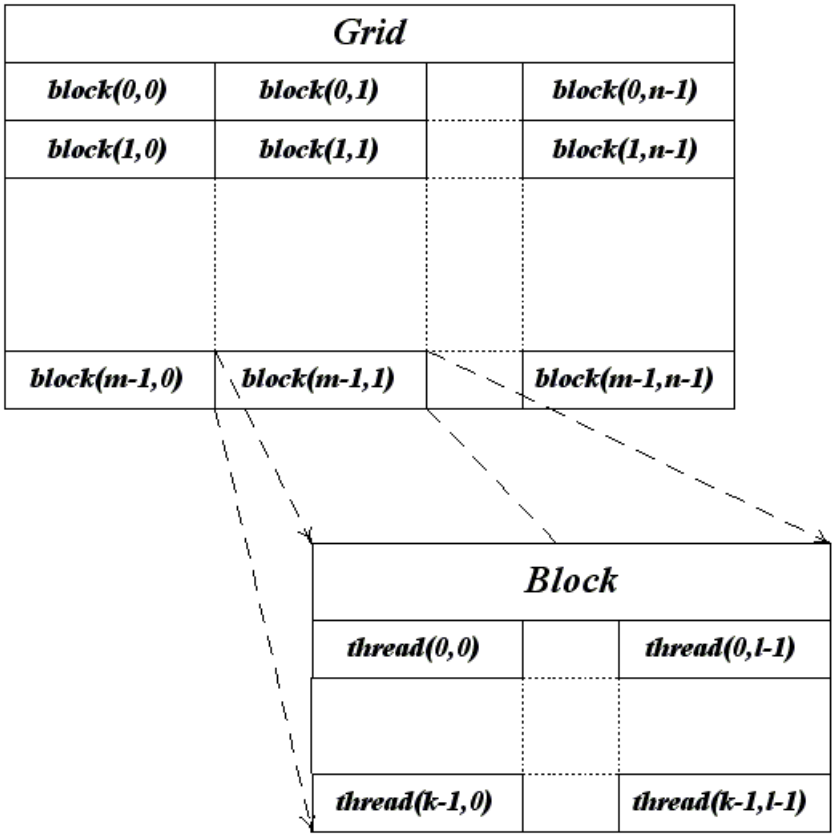

Рисунок 2 – Иерархия нитей в CUDA

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

2.1 Краткий обзор

Механизм трассировки лучей NVIDIA® OptiX™ - программируемая система, разработанная для NVIDIA GPUs и другая очень параллельная архитектура. Механизм OptiX основывается на ключевом наблюдении, что большинство алгоритмов трассировки лучей может быть реализовано, используя маленький набор программируемых операций. Следовательно, ядро OptiX - проблемно-ориентированный своевременный компилятор, который генерирует пользовательские ядра трассировки лучей, комбинируя предоставленные пользователями программы для генерации луча, штриховки материала, объектного перекрестка и обхода сцены. Это включает реализацию очень разнообразного набора основанных на трассировке лучей алгоритмов и приложений, включая интерактивный рендеринг, оффлайн рендеринг, системы обнаружения коллизий, запросы искусственного интеллекта и научные моделирования, такие как звуковое распространение. OptiX достигает высокой производительности через компактную объектную модель и приложение нескольких специфичной для трассировки лучей оптимизации компилятора. Для простоты использования это 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 много-базовой архитектуры, мы 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 неграфических доменах, таких как рендеринг, звуковое распространение, обнаружение коллизий и искусственный интеллект. В этой газете мы обсуждаем цели проекта механизма OptiX, а также реализации для NVIDIA Quadro®, GeForce® и Tesla® GPUs. В нашей реализации мы составляем проблемно-ориентированную компиляцию с гибким набором средств управления иерархией сцены, ускоряющего создания структуры и обхода, непрерывного обновления сцены и динамично сбалансированной с загрузки модели выполнения GPU. Несмотря на то, что OptiX в настоящее время предназначается для очень параллельной архитектуры, это применимо к широкому диапазону специального предложения - и аппаратные средства общего назначения и модели массовой казни.

Чтобы создать систему для широкого диапазона задач трассировки лучей, несколько компромиссов и проектных решений привели к следующим вкладам: • Общий низкоуровневый механизм трассировки лучей. Механизм OptiX фокусируется исключительно на фундаментальных вычислениях, требуемых для трассировки лучей, и избегает встраивать конструкции renderingspecific. Механизм 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, когда объединено с произвольными данными, хранившими с каждым лучом, может использоваться, чтобы реализовать множество сложного рендеринга и нерендеринга алгоритмов. • Простая модель программирования. Механизм OptiX обеспечивает механизмы выполнения, что программисты трассировки лучей приучены к использованию, и избегает обременять пользователя машинным оборудованием высокоэффективных алгоритмов трассировки лучей. Это представляет знакомую рекурсивную, модель программирования единственного луча, а не пакеты луча или явные конструкции SIMD-стиля. Краткие обзоры механизма любая пакетная обработка или переупорядочение лучей, а также алгоритмы для создания высококачественных ускоряющих структур. • 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, levelof-детализируют и вложенные ускоряющие структуры. 2 связанных работы

В то время как многочисленные высокоуровневые библиотеки трассировки лучей, механизмы и ПЧЕЛА были предложены [Вальд и др. 2007b], усилия до настоящего времени фокусировались на определенных приложениях или классах рендеринга алгоритмов, делая их трудными адаптироваться к другим доменам или архитектуре. С другой стороны, несколько исследователей показали, как отобразить алгоритмы трассировки лучей эффективно на GPUs и архитектуру NVIDIA® CUDA™ [Эйла и Лэн 2009; Рожок и др. 2007; Попов и др. 2007], но эти системы фокусировался на производительности, а не гибкости. Основанные на ЦП системы трассировки лучей в реальном времени были сначала разработаны в 1990-х на очень параллельных суперкомпьютерах [Green и Paddon 1990; Muuss 1995; Паркер и др. 1999]. Последующие улучшения ускоряющих структур [Ювелир и Сэлмон 1987; Макдональд и Бут 1989] и пересекающиеся методы [Вальд и др. 2001; Havran 2001; Решетов и др. 2005; Вальд и др. 2007a] включал интерактивную трассировку лучей на машинах настольного класса [Bigler и др. 2006; Георгиев и Слусаллек 2008]. Эти системы были созданы, используя C и/или языки программирования на C++ с традиционными моделями объектно-ориентированного программирования, а не универсальная основанная на программе построения теней система, описанная в этой газете. RPU [Woop и др. 2005] является системой аппаратного обеспечения особого назначения для интерактивной трассировки лучей, которая обеспечивает определенную степень программируемости для использования геометрии, вершины и освещения программ построения теней, записанных в ассемблере. Едкая Графика [Едкая Графика 2009] недавно демонстрировал плату акселератора особого назначения, но не опубликовал детали о механизмах для программирования программ

программы построения теней. OpenRT использовал двоичный сменный интерфейс, чтобы обеспечить поверхность, свет, камеру и программы построения теней среды [Дитрих и др. 2003], но не боролся за общность, предпринятую здесь. Другие интерактивные системы трассировки лучей, такие как Манта [Bigler и др. 2006], Бритва [Djeu и др. 2007], и Arauna [Bikker, 2007] также обеспечивает ПЧЕЛУ, которые являются определенной системой и не предназначенные как решения общего назначения. 3 Программируемый Конвейер Трассировки лучей, которым 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 краткий обзор rasterizer содержащий легкие обратные вызовы для штриховки вершины, обработки геометрии, составления мозаики и операций штриховки фрагмента. Ансамбль этих типов программы, 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 объединенный с определяемой пользователем структурой данных (полезная нагрузка) связался с каждым лучом. Ансамбль программ тайно замышляет реализовывать а алгоритм определенного клиентского приложения. 3.1 Программы Есть семь различных типов программ в OptiX, каждый из которых воздействует на единственный луч за один раз. Кроме того, программа ограничивающего прямоугольника воздействует на геометрию, чтобы определить примитивные границы для ускоряющей конструкции структуры. Комбинация пользовательских программ и кода ядра hardcoded OptiX формирует конвейер трассировки лучей, который обрисован в общих чертах в рисунке 2. В отличие от прямого каналом конвейера растеризации, более естественно думать о конвейере трассировки лучей как о графе вызовов. Базовая работа, rtTrace, чередуется между определением местоположения перекрестка (Пересечение) и ответом на тот перекресток (Оттенок). Читая и при записи данных в определяемых пользователем полезных нагрузках луча и в глобальных матрицах элементов памяти устройства (буферы, посмотрите раздел 3.5), эти операции объединены, чтобы выполнить произвольное вычисление во время трассировки лучей. Программы генерации луча - запись в конвейер трассировки лучей. Единственный вызов rtContextLaunch создаст много инстанцирований этих программ. В примере в рисунке 3 программа генерации луча создаст луч, используя модель камеры с точечной диафрагмой для единственного пикселя, запустить работу трассировки и сохранить получающееся, раскрашивают буфер вывода. С этим механизмом можно также выполнить другие операции, такие как создание карт фотона, вычисления испеченного освещения, обработка запросов луча передавалась от OpenGL, стрельба в многократные лучи для избыточной выборки или реализации различных моделей камеры. Программы перекрестка реализуют тесты на пересечение геометрии луча. Поскольку ускоряющие структуры пересечены, система вызовет программу перекрестка, чтобы выполнить геометрический запрос. Программа определяет, если и где луч касается объекта и может вычислить normals, координаты текстуры или другие атрибуты на основе позиции хита. Произвольное число атрибутов

может быть связано с каждым перекрестком. Программы перекрестка включают поддержку произвольных поверхностей, таких как сферы, цилиндры, старшие поверхности или даже фрактальные конфигурации как компания Джулий в рисунке 1. Однако даже в системе только для треугольника, можно встретиться с большим разнообразием представлений петли. Программируемая работа перекрестка облегчает прямой доступ к собственному формату, который может помочь избегать копий при взаимодействии с основанными на растеризации системами. Программы ограничивающего прямоугольника вычисляют границы, связанные с каждым примитивом, чтобы включить ускоряющие структуры по произвольной геометрии. Учитывая примитивный индекс, простая программа этого типа может, например, считать данные вершины из буфера и вычислить ограничивающий прямоугольник треугольника. Процедурная геометрия может иногда только оценивать границы примитива. Такие оценки позволены, пока они консервативны, но освобождают границы, может ухудшить производительность. Самые близкие программы хита вызваны, как только обход нашел самый близкий перекресток луча с геометрией сцены. Этот тип программы близко напоминает поверхностные программы построения теней в классических системах рендеринга. Как правило, самая близкая программа хита выполнит вычисления как штриховка, потенциально бросая новые лучи в процессе, и хранят данные результата в полезной нагрузке луча. Любые программы хита вызывают во время обхода для каждого объектного лучом перекрестка, который найден. Любая программа хита позволяет материалу участвовать в объектных решениях перекрестка при разделении операций штриховки от операций геометрии. Это может дополнительно завершить луч, используя встроенную функцию rtTerminateRay, который остановит весь обход и раскрутит стек вызовов к новому вызову rtTrace. Это - легкий механизм исключения, который может использоваться, чтобы реализовать раннее завершение луча для теневых лучей и окружающего поглощения газов. Альтернативно, любая программа хита может проигнорировать перекресток, используя rtIgnoreIntersection, позволяя обходу продолжать искать другие геометрические объекты. Перекресток может быть проигнорирован, например, на основе поиска канала текстуры, таким образом реализовывая эффективную отображенную на альфу прозрачность, не перезапуская обход. Другой вариант использования для любой программы хита может быть найден в Разделе 8.1, где приложение выполняет затухание видимости для частичных теней, брошенных стеклянными объектами. Обратите внимание на то, что перекрестки могут быть представлены не в порядке. Значение по умолчанию, которое любая программа хита не, который часто является желаемой работой. Программы мисс выполняются, когда луч не пересекает геометрии в обеспеченном интервале. Они могут использоваться, чтобы реализовать цвет фона или поиск карты среды. Программы исключения выполняются, когда система встречается с исключительным условием, например, когда стек рекурсии превышает объем памяти, доступный для каждого потока, или когда буферный индекс доступа вне диапазона. OptiX также поддерживает определяемые пользователем исключения это может быть брошено из любой программы. Программа исключения может реагировать, например, распечатывая диагностические сообщения или визуализируя условие при записи специальных значений цвета в буфер выходного пикселя. Селекторные программы посещения представляют программируемость для обхода графика узла крупного уровня. Например, приложение может принять решение изменить уровень геометрической детали для

частей сцены на perray основе. В этом случае программа посещения исследовала бы луч расстояние или дифференциал луча, сохраненный полезной нагрузкой и, принимают пересекающееся решение на основе тех данных. 3.2 Представление сцены Механизм OptiX использует гибкую структуру для представления информации о сцене и связал программируемые операции, собранный в контейнерном объекте вызывал контекст. Это представление - также механизм для привязки программируемых программ построения теней к объектно-специфичные данные, которых они требуют. В сочетании с объектной моделью специального назначения, описанной в Разделе 3.3, достигнуто компактное представление данных сцены. 3.2.1 Узлы иерархии Сцена представлена как график. Это представление очень легко и управляет обходом лучей через сцену. Это может также использоваться, чтобы реализовать инстанцирующие двухуровневые иерархии для анимаций твердых объектов или другие общие структуры сцены. Чтобы поддерживать инстанцирование и совместное использование общих данных, у узлов могут быть многократные родители. Четыре основных типов узлов могут использоваться, чтобы обеспечить представление сцены, используя направленный граф. Любой узел может использоваться в качестве корня обхода сцены. Это позволяет, например, различным представлениям использоваться для различных типов луча. Узлы группы содержат нуль или больше (но обычно два или больше) дочерние элементы любого типа узла. Узлу группы связали ускоряющую структуру с ним и может использоваться, чтобы обеспечить верхний уровень двухуровневой пересекающейся структуры. Узлы Geometry Group - листы графика и содержат примитивные и материальные объекты, описанные ниже. Этому типу узла также связали ускоряющую структуру с ним. Любая непустая сцена будет содержать по крайней мере одну группу геометрии. Преобразуйте узлы, имеют единственный дочерний элемент любого типа узла, плюс связанное 4×3 матрица, которая используется, чтобы выполнить аффинное преобразование базовой геометрии. У селекторных узлов есть нуль или больше дочерних элементов любого типа узла плюс единственная программа посещения, которая выполняется, чтобы выбрать среди доступных дочерних элементов. Несмотря на то, что не реализованный в текущей версии библиотек OptiX, график узла может быть циклическим, если селекторный узел используется тщательно, чтобы избежать бесконечной рекурсии. 3.2.2 Геометрия и материальные объекты Объем данных сохранен в узлах геометрии в листах графика. Они содержат объекты, которые определяют операции штриховки и геометрия. У них могут также быть многократные родители, позволяя материалу и информации о геометрии быть совместно использованным в многократных точках в графике; для полного примера посмотрите рисунок 4. Объекты Экземпляра геометрии связывают объект геометрии с рядом материальных объектов. Это - общая структура, используемая графиками сцены, чтобы сохранить геометрическую и заштриховывающую информацию ортогональной. Объекты геометрии содержат список геометрических примитивов. Каждый объект геометрии связан с программой ограничивающего прямоугольника и программой перекрестка, оба из которых совместно использованы среди примитивов объекта геометрии. Материальные объекты содержат информацию о штриховке операций, включая программы призывал к каждому перекрестку, поскольку они обнаружены (любая программа хита) и для перекрестка, самого близкого к источнику данного луча (самая близкая программа хита). 3.3 Объектная модель и модель данных OptiX использует объектную модель специального назначения, разработанную, чтобы

минимизировать постоянные данные, используемые программируемыми операциями. В отличие от системы OpenGL, где только единственная комбинация программ построения теней используется за один раз. Однако трассировка лучей может в произвольном порядке получить доступ к объектным и материальным данным. Поэтому, вместо универсальных переменных, используемых языками штриховки OpenGL, OptiX позволяет любому из объектов и узлов, описанных выше переносить произвольное набор переменных, выраженных как введенная пара значение-имя, вызывал переменную. Переменные установлены клиентским приложением и имеют доступ только для чтения во время выполнения трассировки. Переменные могут иметь скалярные или векторные целочисленные и типы с плавающей точкой (например, float3, int4), а также определяемый пользователем structs и ссылки на буферы и текстурировать сэмплеры. Механизм наследования для этих переменных уникален для OptiX. Вместо основанной на классе модели наследования с синглом сам или этот указатель, механизм OptiX отслеживает текущую геометрию и материальные объекты и текущий пересекающийся узел. Значения переменных наследованы от объектов, которые активны в каждой точке в потоке управления. Например, программа перекрестка наследует определения от геометрия и объекты экземпляра геометрии, в дополнение к глобальным переменным определены в контексте. Концептуально, OptiX исследует каждый из этих объектов для соответствующей пары имя/значение, когда к переменной получают доступ. Этот механизм может считаться обобщением вложенное определение объема найдено в большинстве языков программирования. Это может также быть реализовано вполне эффективно в своевременном компиляторе. Как пример того, как это полезно, полагайте, что массив источников света вызывал световые сигналы. Как правило, пользователь определил бы световые сигналы в контексте, глобальной области видимости OptiX. Это делает это значение доступным во всех программах построения теней во всей сцене. Однако, если световые сигналы для определенная объектная потребность, которая будет переопределена, другая переменная того же имени может быть создана и присоединена к экземпляру геометрии, связанному с тем объектом. Таким образом программы, соединенные с тем объектом, использовали бы переопределенное значение световых сигналов, а не значение, присоединенное к контексту. Это - мощный механизм, который может использоваться, чтобы минимизировать данные сцены, чтобы включить высокую производительность на архитектуре с минимальными кэшами. Способ, которым обработаны эти случаи, может измениться существенно от одного средства рендеринга до другого, таким образом, механизм OptiX обеспечивает основную функциональность, чтобы выразить любого число переопределения управляет эффективно. Специальный тип переменной, тегированной с атрибутом ключевого слова, может использоваться, чтобы передать информацию от программы перекрестка до самого близкого - и любого - программы хита. Они походят на OpenGL переменные переменные и используются для передачи координат текстуры, normals и другой информации о штриховке от программ перекрестка до программ штриховки. У этих переменных есть специальная семантика — они записаны программой перекрестка, но только значения, связанные с самым близким перекрестком, сохранены. Этот механизм позволяет работе перекрестка быть абсолютно отдельной от операций штриховки, включая многократные одновременные примитивы и/или форматы хранения петли при тихой поддержке текстурирования, заштриховывая normals, и объектных искривлений для дифференциалов луча. Атрибуты,

которые не используются никем самым близким - или любым - программа хита, могут игнорироваться компилятором OptiX. 3.4 Динамическая отправка Чтобы позволить многократным операциям трассировки лучей сосуществовать в единственном выполнении, OptiX использует определяемый пользователем тип луча. Тип луча - просто индекс, который выбирает определенный набор слотов для любого хита и самых близких программ хита, которые будут выполняться, когда перекресток найден. Это может использоваться, например, чтобы рассматривать теневые лучи отдельно от другого лучи. Точно так же многократные точки входа в контексте OptiX включают эффективному способу представлять различные передачи по тому же набору геометрии. Например, картопостроитель фотона может использовать одну точку входа, чтобы бросить фотоны в сцену и вторую точку входа, чтобы бросить лучи просмотра. 3.5 Буферы и текстуры Ключевая абстракция для объемного хранения данных - многомерный буферный объект, который представляет 1-, 2-или 3-мерный массив фиксированного размера элемента. К буферу получают доступ через объект обертки C++ в любой из программ. Буферы могут быть только для чтения, только для записи или чтение-запись и поддерживать атомарные операции, когда поддерживается аппаратными средствами. Буфер основан на дескрипторе и не представляет необработанные указатели, таким образом позволяя времени выполнения OptiX переместить буферы для уплотнения хранения, или для продвижения другим пространствам памяти для производительности. Буферы обычно используются для выходных изображений, треугольных данных, списки источника света и другие основанные на массиве данные. Буферы - единственные средства данных вывода из программы OptiX. В большинстве приложений программа генерации луча будет ответственна за запись данных к буферу вывода, но любой из программ OptiX позволяют записать в буферы вывода в любом расположении, но без упорядочивания гарантий. Буфер может также быть связан с объектом сэмплера текстуры, который использует GPU, текстурирующий аппаратные средства. Буферы и объекты сэмплера текстуры связаны с переменными OptiX и используют те же механизмы определения объема как значения программы построения теней. Кроме того, оба буфера и сэмплеры текстуры могут взаимодействовать с OpenGL и DirectX, включая эффективную реализацию гибридных приложений растеризации/трассировки лучей. 4 системных обзора Механизм OptiX состоит из двух отличных ПЧЕЛ, один для стороны узла и один для кода 1 стороны устройства, API узла - ряд C функции, что вызовы клиентского приложения, чтобы создать и сконфигурировать контекст, соберите график узла и запустите ядра трассировки лучей. Это также обеспечивает вызовы, чтобы управлять устройствами, используемыми для выполнения ядра. API программы - функциональность, представленная пользовательским программам. Это включает вызовы функции для трассировки лучей, сообщая о перекрестках, и доступ к данным. Кроме того, несколько семантических переменных кодируют состояние, определенное для трассировки лучей, например, текущее расстояние до самого близкого перекрестка. Печать и средства обработки исключений также доступна для отладки. Рисунок 5 обрисовывает в общих чертах поток управления приложения OptiX. Во время установки приложение вызывает API-функции узла OptiX, чтобы обеспечить данные данных сцены, такие как геометрия, материалы, ускоряющие структуры, иерархические отношения и программы. Последующий вызов к rtContextLaunch API-функции передает управление к OptiX, где изменения в контексте обработаны. При необходимости новое

ядро трассировки лучей скомпилировано из данных пользовательских программ. Ускоряющие структуры созданы (или обновлены), и данные синхронизируются между памятью устройства и узлом. Наконец, ядро трассировки лучей выполняется, вызывая различные пользовательские программы, как описано в Разделе 3. После того, как выполнение ядра трассировки лучей закончилось, его данные результата могут использоваться приложением. Как правило, это включает чтение от буферов вывода, заполненных одной из пользовательских программ или отображения такого буфера непосредственно, например, через OpenGL. Интерактивное или многопроходное приложение тогда повторяет процесс, запускающийся при установке контекста, где произвольные изменения в контексте могут быть внесены, и ядро запущено снова. 5 ускоряющих структур Базовый алгоритм для нахождения перекрестка между лучом и геометрией сцены включает обход ускоряющих структур. Такие структуры данных - жизненный компонент фактически каждой системы трассировки лучей. Они - обычно пространственные или иерархии объектов и используются пересекающимся алгоритмом, чтобы эффективно искать примитивы это потенциально пересекает данный луч. OptiX предлагает гибкий интерфейс, подходящий для широкого диапазона приложений, чтобы управлять его ускоряющими структурами. 5.1 Взаимодействие с графиком узла Одна из причин сбора данных геометрии в графике узла состоит в том, чтобы упростить организацию связанных ускоряющих структур. Вместо того, чтобы поддержать всю геометрию сцены в единственной ускоряющей структуре, это часто целесообразно создавать несколько структур по различным областям сцены. Например, части сцены могут быть анимированы, требуя, чтобы ускоряющая структура была восстановлена для каждой передачи трассировки лучей. В этом случае создание отдельной структуры для статических областей сцены может увеличить эффективность. В дополнение к только построению статической структуры один раз, приложение может обычно инвестировать больший бюджет времени в более высокую качественную сборку. Механизм OptiX связывает ускоряющие структуры со всеми группами и группы геометрии в графике узла. Структуры, присоединенные к группам геометрии, являются низким уровнем, созданным по геометрическим примитивам, которые содержит группа геометрии. Структуры на группах созданы по границам дочерних элементов той группы и таким образом представляют ускоряющие структуры высокого уровня. Эти структуры высокого уровня полезны, чтобы выразить иерархические отношения между геометрией, которая изменена на различных уровнях. Инстанцирование. Важной целью проекта для ускоряющей системы структуры была поддержка гибкого инстанцирования. Здесь, инстанцирование относится к репликации низких издержек геометрии сцены, ссылаясь на те же данные несколько раз, не имея необходимость копировать тяжелые структуры данных. Как описано в Разделе 3.2.1, на узлы в графике можно сослаться многократно, который естественно реализует инстанцирование. Это желательно к не, только делятся информацией геометрии среди экземпляров, но и ускоряющими структурами также. Одновременно, должно быть возможно присвоить данные негеометрии, такие как материальные программы и переменные независимо для каждого экземпляра. Мы принимали решение представить ускоряющие структуры, поскольку отдельный API возражает, что присоединены к группам и группам геометрии. В случае инстанцирования возможно присоединить единственную ускоряющую структуру к многократным узлам, таким образом совместно используя ее данные и избегая

избыточной конструкции той же структуры данных. Метод также приводит к эффективному дополнению и удалению экземпляров во времени выполнения. Рисунок 6 показывает пример графика узла с инстанцированием. Ускоряющие структуры на объединенной геометрии. Деление сцены в многократные ускоряющие структуры уменьшает время сборки структуры, но также и уменьшает производительность обхода луча. В ограничивающем случае полностью статической сцены можно было бы обычно выбирать сингл ускоряющая структура. Одна идея позади ускоряющих структур на группах геометрии состоит в том, чтобы упростить управление данными приложения для того типа установки: вместо того, чтобы иметь необходимость объединить геометрического человека объекты в монолитный блок, они могут остаться организованными как отдельные конфигурации и экземпляры, и легко быть собраны в единственной группе геометрии. Соответствующая ускоряющая структура будет создана по отдельным примитивам любых геометрических объектов, приводящих к максимальной производительности, как будто вся геометрия была объединена. Механизм OptiX будет внутренне заботиться о необходимых бухгалтерских задачах, таких как корректное переотображение существенных индексов. Группа геометрии может также использовать определенную информацию за объект при создании ее ускоряющей структуры. Например, в группе геометрии, содержащей многократные объекты, только единственный, возможно, был изменен между передачами трассировки лучей. OptiX может принять во внимание та информация и опускает некоторые избыточные операции (например. вычисления ограничивающего прямоугольника, посмотрите Раздел 5.3). 5.2 Типы ускоряющих ускоряющих структур Трассировки лучей структур - активная область исследования. Нет никакого единственного типа, который оптимален для всех приложений при всех условиях. Типичный компромисс между различными вариантами - производительность трассировки лучей по сравнению со скоростью конструкции, и у каждого приложения есть различный оптимальный баланс. Поэтому, OptiX обеспечивает много различных ускоряющих типов структуры что приложение может выбрать из. Каждая ускоряющая структура в графике узла может иметь другой тип, позволяя комбинации высококачественных статических структур с динамично обновленными. Большинство типов также подходит для структур высокого уровня, т.е. ускоряющих структур, присоединенных к группам. В настоящее время реализованные ускоряющие структуры включают алгоритмы, фокусируемые на качестве иерархии (например, SBVH [Штих и др. 2009]) на скорости конструкции (например, LBVH [Лаутербах и др. 2009]), и различные промежуточные уровни баланса. 5.3 Конструкция Каждый раз, когда базовая геометрия ускоряющей структуры изменена, например, во время анимации, она явно отмечена для, восстанавливают клиентским приложением. OptiX тогда создает так запланированные ускоряющие структуры на последующем вызове rtContextLaunch API-функции. Первая стадия в ускоряющей конструкции структуры получает ограничивающие прямоугольники геометрии, на которую ссылаются. Это достигнуто, выполняя для каждого геометрического примитива в объекте программу ограничивающего прямоугольника, описанную в Разделе 3.1, который требуется, чтобы возвращать консерватора выровненный осью ограничивающий прямоугольник для его примитивного ввода. Используя эти ограничивающие прямоугольники как элементарные примитивы для ускоряющих структур обеспечивает необходимую абстракцию, чтобы проследить лучи против произвольной определяемой пользователем геометрии (включая несколь-

ко типов геометрии в единственной структуре). Чтобы получить необходимые ограничивающие прямоугольники для высокоуровневых узлов группы в дереве, объединение примитивных ограничивающих прямоугольников сформировано и распространено рекурсивно. Второй этап конструкции состоит из фактического создания требуемых ускоряющих структур, данных полученные ограничивающие прямоугольники. Доступный узел и параллелизм устройства могут быть использованы двумя способами. Во-первых, многократные ускоряющие структуры в графике узла могут быть созданы параллельно, поскольку они независимы. Во-вторых, единственный ускоряющий код сборки структуры может обычно параллелизироваться (см., например, [Шевцов и др. 2007], [Чжоу и др. 2008], [Лаутербах и al.2009]). Заключительные ускоряющие данные структуры помещены в память устройства для потребления кодом обхода луча. 5.4 Настройка В то время как ускоряющие структуры в механизме OptiX разработаны, чтобы выполнить хорошо из поля, иногда необходимо для приложения предоставить дополнительную информацию, чтобы достигнуть максимально возможной производительности. Приложение может поэтому установить ускорение специфичные для структуры свойства, которые влияют на последующие сборки структуры и излучают обходы. Один пример для такого свойства - флаг “ремонта”: если геометрия, используемая ускоряющей структурой BVH, изменилась только немного, часто достаточно просто переоборудовать внутренние ограничивающие прямоугольники BVH вместо того, чтобы восстановить полную структуру с нуля (см. [Лаутербах и др. 2006]). Клиентское приложение может включить это поведение на определенных типах ускоряющих структур, если это предполагает, что получающееся общее время выполнения уменьшится. Такие решения оставляют приложению, поскольку оно обычно обладает контекстной информацией, которая недоступна к OptiX. Процедуры сборки, специализированные к определенным типам геометрических примитивов (в противоположность выровненным осью ограничивающим прямоугольникам, обсужденным выше), являются вторым случаем, где свойства полезны. Приложение может, например, сообщить, что ускорение SBVH structureithat базовая геометрия состоит исключительно из треугольников, и где эти треугольники расположены в памяти. SBVH может тогда выполнить более точный метод построения иерархии, которая приводит к более высокому качеству. 6 проблемно-ориентированных компиляций Ядро времени выполнения узла OptiX - Своевременный (JIT) компилятор, который обеспечивает несколько важных частей функциональности. Во-первых, этап JIT комбинирует все предоставленные пользователями программы программы построения теней в одно или более ядер. Во-вторых, это анализирует график узла, чтобы идентифицировать информационно-зависимую оптимизацию. В-третьих, это обеспечивает зависящий от домена Двоичный интерфейс приложений (ABI) и модель выполнения, которая реализует рекурсию и операции указателя функции на устройстве, которое естественно не поддерживает их. Наконец, получающееся ядро выполняется на GPU, используя API драйвера CUDA. 6.1 Программы OptiX Определенные пользователями программы, часто называемые программой построения теней, обеспечены для API узла OptiX в форме Параллельного Выполнения Потока (PTX) функции [NVIDIA 2010b]. PTX - ассемблер виртуальной машины, который является частью архитектуры CUDA. Это реализует низкоуровневую виртуальную машину, подобную во многих отношениях популярному промежуточному представлению Low-Level Virtual Machine (LLVM) с открытым исходным кодом [Lattner и Adve 2004]. Как LLVM, PTX опреде-

ляет ряд простых инструкций, которые обеспечивают основные операции для арифметики, потока управления и доступа к памяти. PTX также обеспечивает несколько высокоуровневых операций, таких как доступ текстуры и трансцендентальные операции. Также подобный LLVM, PTX принимает бесконечный регистровый файл и краткие обзоры много реальных машинных команд. JIT-компилятор во времени выполнения CUDA выполнит выделение регистра, планирование инструкции, устранение невыполняемого кода и многочисленную другую последнюю оптимизацию, поскольку это производит машинный код, предназначающийся для определенной архитектуры GPU. PTX записан с точки зрения единственного потока и таким образом не требует явных операций манипулирования маской маршрута. Это делает его прямым, чтобы понизить PTX с высокоуровневого языка штриховки при предоставлении времени выполнения OptiX возможности управлять и оптимизировать получающийся код. В то время как это обеспечивает мощный основанный на C++ язык штриховки, это может не быть полезно во всех приложениях. Альтернативно, любой фронтэнд компилятора, который может испустить PTX, мог использоваться. Можно было вообразить фронтэнды для Cg, HLSL, GLSL, MetaSL, OpenSL, RSL, GSL, OpenCL, и т.д., который мог произвести надлежащий PTX для ввода в OptiX. Этим способом OptiX - агностик языка штриховки, так как многократные варианты синтаксиса могли использоваться, чтобы генерировать программы для использования с API во время выполнения. 6.2 PTX к компиляции PTX Учитывая набор функций PTX для определенной сцены, компилятор OptiX переписывает PTX использование многократного PTX к передачам преобразования PTX, которые подобны передачам компилятора, которые оказались успешными в инфраструктуре LLVM. Этим способом OptiX использует PTX в качестве промежуточного представления, а не традиционной системы команд. Этот процесс реализует много проблемно-ориентированные операции включая ABI (вызывающая последовательность), разовая ссылкой оптимизация и информационно-зависимая оптимизация. Факт, что большинство структур данных в типичном трассировщике лучей только для чтения, обеспечивает существенную возможность для оптимизации, которую не считали бы безопасной в более общей среде. Анализ. Первая стадия этого процесса должна выполнить статический анализ всех обеспеченных функций PTX. Эта передача гарантирует, что переменные, на которые ссылаются в каждой функции, были обеспечены графиком узла и имеют непротиворечивые типы. Одновременно, мы определяем, только для чтения ли каждый из буферов данных или чтение-запись, чтобы сообщить времени выполнения, где данные должны храниться. Наконец, эта передача может проанализировать структуру графика узла в подготовке к другой специфичной для данных оптимизации, показанной ниже. Встройте instrinsic операции. Время выполнения OptiX обеспечивает несколько операций вне тех предоставленных CUDA. Эти инструкции заменены встроенной функцией, которая реализует требуемые операции. Примеры включают доступ к в настоящее время активному источнику луча, направлению и полезной нагрузке, абстракции хранилища поверхности записи чтения и доступу к стеку преобразования. Кроме того, мы обрабатываем псевдо инструкции, соответствующие исключительному потоку управления, такие как rtTerminateRay и rtIgnoreIntersection. Объектная модель переменной программы построения теней. Программа может сослаться на a переменная программы построения теней без дополнительного синтаксиса, так же, как к задействованной переменной получили бы доступ

в C++. Эти доступы проявят в PTX как инструкция загрузки, связанная со специально теговыми глобальными переменными. Мы обнаруживаем доступы к этим переменным, используя анализ потока данных, передают и заменяют их загрузкой, индексированной от указателя до текущей геометрии, материала, экземпляра или другого объекта API, как определено аналитической передачей. Чтобы реализовать динамическое наследование переменных, маленькая таблица, связанная с каждым объектом, определяет указатель базы и связанное смещение. Продолжения. Считайте программу программы построения теней показанной в рисунке 7 и соответствующем PTX показанный в рисунке 8. Эта программа реализует простой цикл, чтобы проследить 5 лучей от точек (0,0,0), (1,0,0), (2,0,0), (3,0,0) и (4,0,0). В то время как не полезная программа, этот пример может использоваться, чтобы проиллюстрировать, как используются продолжения. Чтобы позволить этому циклу выполняться как ожидалось, переменная, я должен быть спасен прежде временно отказываясь от выполнения этой программы, чтобы вызвать функцию rtTrace. Это выполнено, реализовывая отсталую аналитическую передачу потока данных, чтобы определить регистры PTX, которые живы, когда с псевдоинструкцией для rtTrace встречаются. Живой регистр - тот, который используется в качестве параметра за некоторую последующую инструкцию в графе потоков данных. Мы резервируем слоты на стеке для каждой из этих переменных, упаковываем их в 16-байтовые векторы, если это возможно, и храним их на стеке перед вызовом и восстанавливаем их после вызова. Это подобно ABI сохранения вызывающая сторона, который традиционный компилятор реализовал бы для основанного на ЦП языка программирования. В подготовке к представлению продолжений мы выполняем поднимающую цикл передачу, и распространение копии передают каждую функцию, чтобы помочь минимизировать состояние, сохраненное в каждом продолжении. Наконец, rtTrace псевдоинструкция заменена ответвлением, чтобы возвратить выполнение конечному автомату, описанному ниже, и метка, которая может использоваться, чтобы в конечном счете возвратить поток управления этой функции. Это преобразование приводит к псевдокоду, показанному в рисунке 9. Однако неструктурный gotos в этом коде приведет к неприводимому графику потока управления из-за ввода цикла и наверху цикла и наверху метки state2. Неприводимый поток управления мешает механизмам в GPU, чтобы управлять выполнением SIMD этой функции, приводящей к драматическому замедлению для расходящегося кода. Следовательно, мы разделяем эту функцию, клонируя узлы в графике для каждого состояния. После выполнения устранения невыполняемого кода получена кодовая последовательность в рисунке 10. Этот поток управления более дружественный по отношению к выполнению SIMD, потому что это хорошо структурировано. Расхождение может быть далее уменьшено, представляя новые состояния вокруг общего кода. Это заключительное преобразование может или может не стоить, в зависимости от стоимости переключения состояний и степени расхождения выполнения. 6.3 Оптимизация Инфраструктура компилятора OptiX обеспечивает ряд проблемно-ориентированной и информационно-зависимой оптимизации, которая была бы сложна, чтобы реализовать статически скомпилированную среду. Они включают (увеличения производительности для множества приложений в круглых скобках): • Игнорируйте операции преобразования для графиков узла, которые не используют узел преобразования (до 7%-го повышения производительности). • Устраните печать и исключение связанный код, если эти опции не включены

в текущем выполнении. • Уменьшите размер продолжения, регенерируя константы и промежуточные звенья после восстановления. Так как модель выполнения OptiX гарантирует, что объектно-специфичные переменные только для чтения, эта локальная оптимизация не требует межпроцедурной передачи. • Специализируйте обход на основе древовидных характеристик, таких как существование вырожденных листов, вырожденных деревьев, совместно используемых ускоряющих данных структуры, или смешал типы примитивов. • Переместите маленькие данные только для чтения в постоянную память или текстуры, если есть свободное место (до 29%-го повышения производительности). Кроме того, передачи перезаписи могут представить существенные модификации коду, который может быть очищен дополнительными стандартными оптимизационными проходами, такими как устранение невыполняемого кода, постоянное распространение, подъем цикла и распространение копии. 7 моделей выполнения Различные авторы предложили различные модели выполнения для параллельной трассировки лучей. В частности монолитное ядро или мегаядро, подход оказывается успешным на современном GPUs [Эйла и Лэн 2009]. Этот подход минимизирует запуск ядра наверху, но потенциально уменьшает использование процессора, когда требования регистра растут к максимуму через составляющие ядра. Поскольку задержка при обращении к памяти маски GPUs с многопоточностью, это - тонкий компромисс. OptiX реализует мегаядро, соединяя ряд отдельных пользовательских программ и пересекая конечный автомат, вызванный потоком выполнения между ними во времени выполнения. Поскольку GPUs развиваются, различные модели выполнения могут стать практичными. Например, модель выполнения потоковой передачи [Gribble и Ramani 2008] может быть полезной на некоторой архитектуре. Другая архитектура может обеспечить поддержку аппаратных средств для ускоряющего обхода структуры или других общих операций. Так как механизм OptiX не предписывает, чтобы порядок выполнения между корнями деревьев луча, эти альтернативы могли быть предназначены с передачей перезаписи, подобной той, мы в настоящее время используем, чтобы генерировать мегаядро. 7.1 Выполнение мегаядра Прямой подход к выполнению мегаядра - простая итерация по конструкции случая переключателя. В каждом случае выполняется пользовательская программа, и результат этого вычисления имеет место, или состояние, чтобы выбрать на следующей итерации. В таком механизме конечного автомата OptiX может реализовать вызовы функции, рекурсию и исключения. Рисунок 11 иллюстрирует простой конечный автомат. Состояния программы просто вставлены в тело оператора переключения. Государственный индекс, который мы вызываем виртуальный счетчик команд (VPC), выбирает отрывок программы, который будет выполняться затем. Вызовы функции реализованы устанавливая VPC непосредственно, вызовы виртуальной функции реализованный, устанавливая его от таблицы и функциональных возвратов просто восстанавливают состояние к продолжению, связанному с ранее активной функцией (виртуальный обратный адрес). Кроме того, специальный поток управления, такой как исключения управляет VPC непосредственно, создавая переход требуемого состояния способом, подобным легкой версии setjmp / longjmp функциональность, обеспеченная C. 7.2 Мелкомодульное планирование В то время как прямой подход к выполнению мегаядра функционально корректен, это переносит штрафы сериализации, когда состояние отличает в единственном модуле SIMT [Линдхольма и др. 2008]. Чтобы смягчить эффекты расхождения выполнения, время выполнения OptiX использует мелкомо-

дульную схему планирования исправить расходящиеся потоки это было бы, иначе бездействовал. Вместо того, чтобы позволить аппаратным средствам SIMT автоматически сериализировать выполнение расходящегося переключателя, OptiX явно выбирает единственное состояние для всего модуля SIMT, чтобы выполнить использование эвристики планирования. Потоки в модуле SIMT, которые не требуют состояния просто, бездействуют та итерация. Механизм обрисован в общих чертах в рисунке 12. Мы экспериментировали со множеством мелкомодульной эвристики планирования. Одна простая схема, которая работает хорошо, определяет расписание, присваивая статическое установление приоритетов по состояниям. Планируя потоки с подобными состояниями во время выполнения, OptiX сокращает количество общих изменений состояния, сделанных модулем SIMT, который может существенно уменьшите время выполнения по автоматическому расписанию, вызванному аппаратными средствами сериализации. Рисунок 13 показывает пример такого сокращения. 7.3 Выравнивание нагрузки В дополнение к уменьшению расхождения выполнения SIMT с мелкомодульным планировщиком OptiX использует трехярусный подход балансирования динамической нагрузки к GPUs. Каждый запуск ядра трассировки лучей представлен как очередь задач параллели данных к физическим модулям выполнения. Текущая модель выполнения осуществляет независимость между эти задачи, позволяя балансировщику загрузки динамично запланировать работу на основе характеристик рабочей нагрузки и аппаратных средств выполнения. Работа распределена от узла ЦП до одного или более GPUs динамично, чтобы включить крупномодульное выравнивание нагрузки между GPUs отличающейся производительности. Как только пакет работы был представлен GPU, это помещено в глобальную очередь. Каждый модуль выполнения на GPU присвоен локальная очередь, которая переполнена от глобальной очереди GPU и динамично распределяет работу отдельным элементам обработки, когда они завершили свое текущее задание. Это - расширение схемы, используемой [Эйла и Лэн 2009], чтобы включить динамическую нагрузку, балансирующуюся между GPUs. 8 тематических исследований приложения Этот раздел представляет различные варианты использования OptiX, обсуждая основные идеи позади многих различных приложений. 8.1 Whitted-разработайте трассировку лучей, OptiX SDK содержит несколько приложений трассировки лучей в качестве примера. Один из них - обновленное воссоздание исходной сцены сферы Виттеда [1980]. Эта сцена проста, все же демонстрирует важные функции механизма OptiX. Программа генерации луча выборки реализует основную модель камеры с точечной диафрагмой. Позиция камеры, ориентация и видимое пространство определены рядом переменных программы, которые могут быть изменены в интерактивном режиме. Программа генерации луча начинает процесс штриховки, стреляя в единственный луч за пиксель или, по пользовательскому запросу, выполняя адаптивное сглаживание через избыточную выборку. Материальные самые близкие программы хита тогда ответственны за то, что рекурсивно бросили лучи и вычислили теневой демонстрационный цвет. После возврата из рекурсии программа генерации луча накапливает демонстрационный цвет, сохраненный в полезной нагрузке луча, в буфер вывода. Приложение определяет трех отдельных пар перекрестка и программ ограничивающего прямоугольника, каждый реализующий различный геометрический примитив: параллелограм для пола, сфера для металлического шара и сфера тонкой оболочки для полого стеклянного шара. Стеклянный шар, возможно, был смоделирован

с двумя экземплярами простой примитивной сферы, но гибкость модели программы OptiX дает нам свободу реализовать более эффективную специализированную версию для этого случая. Каждая программа перекрестка устанавливает несколько переменных атрибута: геометрическое нормальное, нормальная штриховка, и, при необходимости координата текстуры. Атрибуты используются материальными программами, чтобы выполнить вычисления штриховки. Механизм типа луча используется, чтобы дифференцировать сияние от теневых лучей. Приложение присоединяет тривиальную программу, которая сразу завершает луч к любым слотам хита материалов для теневых лучей. Это раннее завершение луча приводит к высокой эффективности для взаимных тестов видимости между точкой штриховки и источником света. стеклянный материал - исключение, однако: здесь, любая программа хита используется, чтобы ослабить фактор видимости, сохраненный в полезной нагрузке луча. В результате стеклянная сфера бросает более тонкую тень, чем металлическая сфера. 8.2 Гараж проекта NVIDIA Гараж Проекта NVIDIA - сложная интерактивная демонстрация рендеринга, предназначенная для общедоступного распределения. Главное изображение рисунка 1 было представлено, используя это программное обеспечение. Ядро Гаража Проекта - физическая система трассировки пути Монте-Карло [Kajiya 1986], что постоянно демонстрационные световые пути и совершенствовали изображение оценка, интегрируя новые выборки в течение долгого времени. Пользователь может в интерактивном режиме просмотреть и отредактировать сцену, поскольку начальное шумное изображение сходится к окончательному решению. Чтобы управлять использованием стека, Гараж Проекта реализует трассировку пути, используя итерацию в программе генерации луча вместо того, чтобы рекурсивно вызвать rtTrace. Псевдокод рисунка 15 подводит итог. В Гараже Проекта каждый материал использует самую близкую программу хита, чтобы определить следующий луч, который будет прослежен и пасует назад это использование определенного поля в полезной нагрузке луча. Самая близкая программа хита также вычисляет пропускную способность текущего легкого возврата, который используется генерация луча, чтобы поддержать кумулятивный продукт пропускной способности по полному световому пути. Умножение цвета источника света, пораженного последним лучом по пути, приводит к заключительному демонстрационному вкладу. Поддержка OptiX C++ в программах луча позволяет материалам совместно использовать универсальную самую близкую реализацию хита, параметризованную на тип BSDF. Это позволяет нам реализовывать новые материалы как классы BSDF с методами для выборки важности, а также BSDF и вероятности оценка плотности. Гараж проекта реализует число из различных физических материалов, включая металлическую и автомобильную краску. Некоторые из этих программ построения теней поддерживают нормальные и зеркальные карты. В то время как OptiX реализует всю функциональность трассировки лучей Гаража Проекта, конвейер OpenGL реализует заключительную реконструкцию изображения и дисплей. Этот конвейер выполняет различные этапы обработки сообщения, такие как тональное отображение, блик и фильтрация стандарта использования основанные на растеризации методы. 8.3 Отображение фотона пространства изображения Image Space Photon Mapping (ISPM) [Макгуайр и Луебк, 2009] является алгоритмом рендеринга в реальном времени, который комбинирует стратегии трассировки лучей и растеризации (рисунок 16). Мы портировали опубликованную реализацию на механизм OptiX. Тот процесс дает понимание различий между традиционным

векторизованным последовательным трассировщиком лучей и OptiX. Алгоритм ISPM вычисляет первый сегмент путей фотона от света, растеризируя “карту возврата” от ссылочного фрейма света. Это тогда распространяет фотоны трассировкой лучей с Русской рулеткой, выбирающей до последнего события рассеивания перед глазом. В каждом событии рассеивания фотон депонирован в массив, который является “картой фотона”. Косвенное освещение тогда собрано в изображении пространство, растеризируя небольшой объем вокруг каждого фотона с точки зрения глаза. Прямое освещение вычислено схемами затенения и растеризацией. Рассмотрите структуру трассировщика фотона ЦП-ISPM. Это запускает один персистентный поток за ядро. Эти потоки обрабатывают пути фотона от глобальной переменной, незапертого рабочего списка. Отображение фотона ISPM генерирует несвязные лучи, таким образом, традиционные пакетные стратегии векторизации обхода луча не помогают с этим процессом. Для каждого пути поток обработки вводит цикл с условием продолжения, внося один фотон в глобальной переменной, незапертом массиве фотона за итерацию. Цикл завершается после поглощения фотонов. Под OptiX-ISPM мы также поддерживаем глобальные незапертые буферы ввода и вывода. Увеличения производительности трассировки с успехом мелкомодульного планирования программ в когерентные модули SIMT и уменьшения с размером состояния связывались между программами. Имитация традиционного стиля ЦП архитектуры программного обеспечения была бы неэффективна под OptiX, потому что это потребует передачи всех материальных параметров между генерацией луча и поразит программы и переменный итеративный цикл с условием продолжения в самой близкой программе хита. OptiXISPM поэтому следует альтернативному проекту, который рассматривает все распространение итерации как сопрограммы. Это содержит единственную программу генерации луча с одним потоком за путь фотона. Рекурсивная самая близкая программа хита реализует итерации распространять-и-вносить. Это позволяет потокам уступать между итерациями так, чтобы мелкомодульный планировщик мог перегруппировать их. Мы отмечаем, что широкий подход, проявленный здесь, является способом объединить API растровой графики как OpenGL или DirectX с примитивами трассировки лучей, не расширяя растровый API. Задержанная штриховка - форма получения, где буферы геометрии походят на functionalprogramming продолжение, которое содержит состояние прерванного пиксельный шейдер. Те буферы рассматривает как ввод OptiX API. Это выписывает результаты к другому буферу, и мы тогда эффективно возобновляем процесс штриховки, представляя объемы по буферам геометрии с новым пиксельным шейдером. 8.4 Обнаружение коллизий OptiX предназначен, чтобы быть полезным для нерендеринга приложений также. Центральная панель в рисунке 1 показывает визуализацию OpenGL от обнаружения коллизий, и механизм угла обзора основывался на механизме OptiX. В этом примере механизм моделирует 4096 движущихся объектов, прослеживая лучи против статических 1.1 миллионов сцен многоугольника. Механизм прослеживает 512 лучей зонда коллизии от каждого объектного центра, используя самую близкую программу хита и 40962/2 лучи угла обзора между всеми парами объектов, используя любую программу хита. Включая время, чтобы обработать результаты коллизии и выполнить объектную динамику, механизм достигает 25 миллионов лучей/секунда на GeForce GTX 280 и 48 миллионах лучи в секунду на GTX 480. В то время как луч, бросая approch не устойчиво ко всем операциям коллизии, это - часто используемый метод из-за своей

простоты. 9 результатов производительности Все результаты в этом разделе были представлены в HD 1080 пунктами (1920×1080) разрешение. Чтобы оценить основную производительность, достижимую ядрами OptiX, мы воссоздали некоторые эксперименты, выполняемые в [Эйла и Лэн 2009] использование тех же сцен и позиций камеры. Мы сравненный наши сгенерированные ядра с этими вручную оптимизированными ядрами, чтобы измерить издержки, создаваемые уровнями абстракции программного обеспечения. Мы измеряли необработанную трассировку лучей и времена перекрестка, игнорируя времена для установки сцены, компиляции ядра, ускоряющих сборок структуры, буферных передач, и т.д. Эквивалентные ускоряющие структуры и рассчитывающие механизмы использовались в обеих системах. Таблица 1 показывает, что результаты для работают на NVIDIA GeForce GTX 285 и GeForceGTX 480 GPUs, усредненном по тем же 5 точкам зрения, используемым в исходной газете. В то время как, как ожидалось, гибкость и программируемость OptiX дорого обходится, разрыв производительности все еще приемлем. Самый большой разрыв существует для окружающих лучей поглощения газов, который является частично из-за остающегося недостатка в сравнительном тесте. В частности мы не выполняли сортировку луча и использовали более низкое число вторичных лучей за пиксель для наших измерений. Таблица 2 показывает показатели производительности для Гаража Проекта (см. Раздел 8.2) на NVIDIA Quadro FX5800 и GeForce GTX 480 GPUs, который более показателен из реальной сцены, чем вышеупомянутый тест. Это приложение сложно по нескольким причинам. Во-первых, это - физический код трассировки пути с выборкой комплекса, многократными материалами и многими другими функциями. Это приводит к большому ядру, которое требует многих регистров, таким образом сокращая количество потоков, которые могут работать параллельно на GPU. Во-вторых, потоки более вероятно отличаться рано должный рассеяться или глянцевые легкие возвраты, которые приводят к различным материальным выполняемым программам построения теней, вызывая, уменьшало эффективность SIMT. В-третьих, подразделение геометрии сцены в многократные ускоряющие структуры (чтобы поддерживать анимацию) дополнительно увеличивает число операций для обхода луча по сравнению с монолитная структура данных. Тем не менее механизм OptiX может успешно объединить все эти различные программы и все еще сделать Гараж Проекта достаточно быстро, чтобы предложить интерактивную модификацию сцены и сходимости к фотореалистическому изображению в течение секунд. Мы также сравнили реализацию OptiX-ISPM с опубликованной реализацией ЦП на компьютере Core 2 Quad Intel с рендерингом GTX485 GPU в разрешении HD 1080 пунктов. Мы оценили 20 сцен, включая "атриум Sponza" и "NS2" [Макгуайр и Луебк 2009]. Таблица 3 суммирует результаты производительности для четырех представительных сцен. Все были представлены с 4×4 подвыборка в глобальный сборочный шаг. Локальное время освещения включает буферы геометрии и схемы затенения. Время ввода-вывода измеряет передачу данных между OpenGL и ЦП или памятью CUDA. Сетевое время Локально + Глобальная переменная + Трассировка + ввод-вывод. Типичное ускорение было о 4× для трассировки и 2.5× в целом. "NS2" привел к самому низкому сетевому ускорению, с фотон OptiX прослеживает 3.0× быстрее, чем ЦП один и сетевое время 1.8× быстрее. Обратите внимание на то, что нахождение на той же стороне шины PCI так же важно как вычислительная производительность. Предотвращение передачи данных GPU ЦП может уменьшить время ввода-вывода на целых 50%. По-

вышение эффективности обмена данными между двумя ПЧЕЛАМИ далее уменьшит стоимость этой передачи данных. 10 ограничений и будущая работа В настоящее время операции двойной точности поддержки во время выполнения OptiX в программах, но лучи сохранены в одинарной точности. Для некоторых приложений было бы желательно иметь луч двойной точности. Расширения буферного механизма OptiX подали бы некоторые проще заявки, такие как операции для добавляют, сокращения и сортирующие значения. В некотором applicatons может также требоваться механизм динамического выделения памяти. Как с большинством компиляторов, есть бесконечные возможности для дополнительных оптимизационных проходов, которые будут добавлены, поскольку мы приобретаем опыт с системой на важных приложениях. Кроме того, было бы интересно видеть, что дополнительные языки штриховки предназначаются для OptiX через PTX. Ускоряющие структуры OptiX созданы, используя программу ограничивающего прямоугольника или специальный API, который поддерживает только треугольные данные. Чтобы создать лучшие ускоряющие структуры для программируемой геометрии, это было бы выгодно, чтобы обобщить ускоряющие сборки структуры, чтобы позволить дополнительные программируемые операции. Это могло бы включать определяемое пользователем поле / примитивный тест перекрытия среди других операций. OptiX поддерживает несколько типов ускоряющих структур, но в настоящее время не предоставляет механизм пользователю, чтобы реализовать их собственное. 11 Заключений 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. OptiX связывает эти программы с узлами в графике, который определяет геометрическую конфигурацию и ускоряющие структуры данных, против которых прослежены лучи. Наши вклады включают низкоуровневый API трассировки лучей и связанную модель программирования, понятие программируемого конвейер трассировки лучей и связанный набор программы типы, проблемно-ориентированный JIT-компилятор, который выполняет преобразования мегаядра и реализует несколько проблемно-ориентированной оптимизации и легкое представление сцены, которое предоставляет себя высокоэффективной трассировке лучей и поддержкам, но не ограничивает, структура графика сцены приложения. Механизм трассировки лучей OptiX - поставляющий продукт и уже поддерживает широкий диапазон из приложений. Мы иллюстрируем широкую применимость OptiX с многократными примерами в пределах от упрощенного к довольно сложному. Подтверждения Автомобиль, лягушка и модель механизма в рисунке 1 - любезность TurboSquid. Модель кролика 16 в цифрах и 4 является любезностью Stanford University Graphics Lab. Фил Миллер способствовал хранению усилия на ходу. Авторы ценят ценные комментарии от доктора Грега Хумфреиса и извлекли выгоду значительно из основы и многочисленных переговоров на трассировке лучей с элементами из Исследования NVIDIA и команды SceniX.

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. Г.Б. Усынин, Е.В. Кусмарцев. Реакторы на быстрых нейтронах: Учеб. пособие для вузов/Под ред. Ф.М. Митенкова. — М.: Энергоатомиздат, 1985. — 288 с.: ил.

4. Официальный сайт ОАО "ННЦ НИИАР" (www.niiar.ru)

5. Джон Хопкрофт, Раджив Мотвани, Джеффри Ульман Введение в теорию автоматов, языков и вычислений = Introduction to Automata Theory, Languages, and Computation. — М.: Вильямс, 2002. — 528 с. ISBN 0-201-44124-1

6. В.И. Носов, Т.В. Бернштейн, Н.В. Носкова, Т.В. Храмова. Элементы теории графов. Учебное пособие. — Новосибирск, 2008. — 107 с.

7. С.Дж. Рассел, П. Норвиг. Искусственный интеллект: современный подход = Artificial Intelligence: A Modern Approach / Пер. с англ. и ред. К. А. Птицына. — 2-е изд.. — М.: Вильямс, 2006. — 157—162 сс. ISBN 5-8459-0887-6

8. Д. Восквитцов. Реализация графов и деревьев на Python

9. Гвидо ван Россум, Шаблоны Python — реализация графов = Python Patterns - Implementing Graphs / Пер. с англ. С.Тезадов.

10. Официальный сайт проекта NetworkX (http://networkx.lanl.gov/reference/index.html)