

Смещённое решение интегрального уравнения светопереноса на графических процессорах при помощи трассировки путей и кэша освещенности

Владимир Фролов^(1,2), Александр Харламов^(1,2) и Алексей Игнатенко⁽¹⁾

(1) Факультет Вычислительной Математики и Кибернетики, Московский государственный университет имени М.В. Ломоносова. {vfrolov, ignatenko}@graphics.cs.msu.ru

(2) NVIDIA, akharlamov@nvidia.com

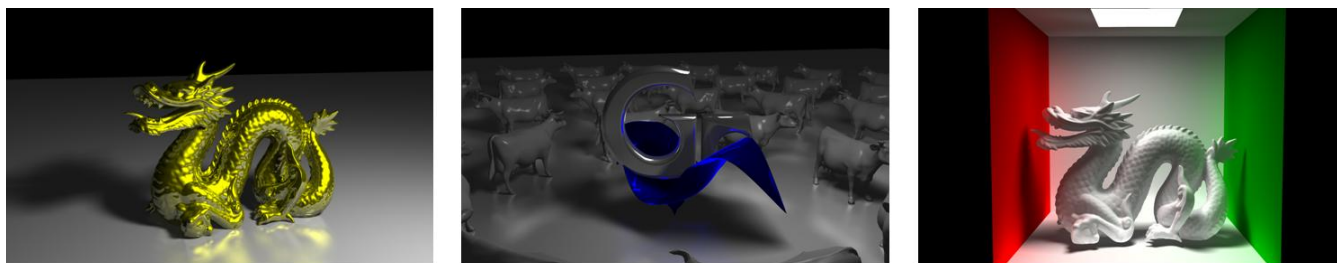


Рис. 1. Представленный гибридный подход использует кэш освещенности для аппроксимации гладкого вторичного освещения и трассировку путей для расчета мягких теней, размытых отражений и других эффектов. Все изображения были синтезированы в разрешении 1920x1200 на графическом процессоре GTX 260 менее чем за 3 минуты.

1. АННОТАЦИЯ

В данной работе представлен подход к синтезу фотореалистичных изображений на графических процессорах (GPU). В его основе лежит сочетание алгоритмов кэширования освещенности с когерентной адаптивной трассировкой путей.

Key words: *Global illumination, photorealistic rendering, GPU computing.*

2. ВВЕДЕНИЕ

В течение последних десяти лет архитектура графических процессоров сильно изменилась. В начале фиксированный конвейер заменили программируемым, затем исчезло разделение на вершинные и фрагментные процессоры, появились программные модели и средства разработки для вычислительных приложений, никак не связанных с тем, что GPU были призваны ускорять изначально – растеризацией. С появлением новых программных моделей возрос интерес к использованию GPU для трассировки лучей. В нашей работе мы используем технологию CUDA. Однако в виду того, что программная модель CUDA аналогична таким моделям API как OpenCL и DirectX, мы можем обобщить наши результаты и сделать выводы, независимые от аппаратного обеспечения.

В настоящее время доступен ряд программных пакетов для фотореалистичного синтеза изображений на GPU (IRay, Octane, Arion), использующих методы, дающие так называемое несмещенное решение. Однако у таких подходов есть два существенных ограничения. Во-первых, улучшение качества результата достигается за счет сильного увеличения алгоритмической сложности, и как следствие, увеличения объемов вычислений. На практике такие методы могут использовать на порядок больше лучей, чем альтернативные подходы (такие как метод кэширования освещенности). Вторая проблема кроется в сильной нерегулярности самого процесса трассировки лучей. Несмотря на то, что каждый луч

может быть обработан независимо, общая вычислительная нагрузка и обращения к памяти сильно нерегулярны. Это создает дополнительные трудности при реализации трассировки лучей на массивно-параллельном процессоре. На практике более эффективны, алгоритмы, дающие смещённое решение. Однако их реализация на массивно-параллельных системах вызывает серьезные трудности из-за ещё более сильной неравномерности распределения вычислительной нагрузки и рекурсивной природы таких алгоритмов. В данной работе рассмотрена эффективная реализация трассировки лучей на графических процессорах, а также представлены методы получения фотореалистичных изображений с помощью алгоритма кэширования освещенности для расчета вторичного освещения и алгоритма трассировки путей для быстрого расчета мягких теней, размытых отражений и некоторых других эффектов.

3. ПРЕДЫДУЩИЕ МЕТОДЫ

3.1 Трассировка лучей на GPU

В работе [1] предложено использовать традиционный графический конвейер. Авторы [1] реализовали алгоритм трассировки лучей в виде набора фрагментных программ. В качестве ускоряющей структуры данных использовалась регулярная сетка. Конвейер данных был организован следующим образом: луч генерировался в первом ядре. Второе ядро выполняло поиск в ускоряющей структуре данных (регулярной сетке). Когда луч попадал в воксель, содержащий треугольники, он передавался следующему ядру, которое тестировало луч на пересечение с треугольниками. Процесс продолжался до тех пор, пока луч не находил пересечение или не покидал трассируемый объем. Управление состоянием (обход сетки, пересечение с примитивами, вычисление цвета) луча осуществлялось с помощью буфера трафарета. При таком конвейере проще осуществлять отладку и можно более сфокусировано

анализировать производительность и узкие места приложения.

В работе [2] предложили 2 альтернативных подхода для трассировки лучей с использованием Kd-дерева. GPU не поддерживают стек, поэтому предложено было реализовать один из двух возможных вариантов:

- 1) Изменить структуру узла Kd-дерева так, чтобы она поддерживала ссылку на родительский узел. Эта ссылка используется каждый раз, когда лучу необходимо вернуться на верхний уровень иерархии и продолжить обход другого поддерева.
- 2) Проводить обход дерева, пока не будет найден непустой лист. Если, однако, в этом непустом листе пересечения луча с примитивом не произошло, то начало луча сдвигается таким образом, чтобы данный лист, при повторном обходе дерева измененным лучом был пропущен. Процесс обхода дерева начинается заново с корня для измененного луча и продолжается, пока не найдено пересечение или пока луч не покинет рассматриваемый объем.

В работе [3] предложено видоизменить второй алгоритм из [2] так, чтобы хранить в регистрах короткий стек. Это позволяет избежать слишком частого обхода дерева.

Для BVH (Иерархии ограничивающих объемов) в [4] был представлен бесстековый алгоритм и эффективная реализация трассировки на GPU. Каждый лист в таком дереве хранит дополнительную ссылку, как показано на рисунке 2. Эта ссылка указывает на то поддерево, которое необходимо обойти в случае, если тест на пересечение луча с примитивами в узле провалится.

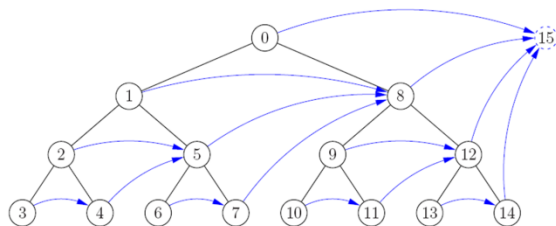


Рис. 2 BVH structure for stackless traversal.

3.2 Глобальное освещение на GPU

В работе [5] представлен эффективный подход к расчету глобального освещения на GPU, основанный на использовании фотонных карт. Основной идеей этой работы стало использование кэша освещенности и процедуры финального сбора, для быстрого расчета вторичного освещения. Для расчёта первичного освещения были использованы теневые карты и обратная трассировка лучей.

В работе [5] положение точек для кэша освещенности определялось исходя из геометрических разрывов. Для адаптивного разбиения при поиске положений было использовано квадро-дерево. Для оценки геометрических разрывов (разрывы поля нормалей и позиций в пространстве экрана) была введена следующая метрика: если расстояние между угловыми точками в узле квадро-дерева превышает некоторую наперед заданную ошибку, то разбиение данного узла продолжается, в противном случае, дополнительного

разбиения не требуется. Похожий подход был описан в [6], но без использования процедуры финального сбора. Прямое освещение было рассчитано используя трассировку лучей, вторичное освещение – с помощью фотонных карт. Такой алгоритм хорошо работает с каустиками, но для вторичного освещения результат получается шумным.

В [7] фотонные карты использовались для расчета интеграла освещенности напрямую (и первичное и вторичное освещение вычислялось с помощью этого метода). Недостатком такого решения служит сильный низкочастотный шум и темные границы. Низкочастотный шум – проблема метода фотонных карт, которую обычно решают с помощью процедуры финального сбора или фильтрации в пространстве объекта. Темные края на границах геометрии возникают от того, что освещение складывается только с половины диска, а результирующее значение получается делением на всю площадь диска.

Фильтрация и финальный сбор вносят дополнительное смещение в решение интегрального уравнения. К тому же у финального сбора возникают проблемы, когда две поверхности расположены близко и на них не попадают фотоны. Чтобы избавиться от этих артефактов, можно использовать более медленный вторичный проход финального сбора (как описано в [8]).

В работе [9] была использована растеризация для первичного освещения в сочетании с трассировкой фотонов на CPU и последующей растеризацией фотонов на GPU для аппроксимации вторичного освещения.

Таким образом, методы описанные в [5], [6] и [9] могут быть использованы в интерактивных приложениях, но они не дают фотореалистичного изображения.

4. АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ ТРАССИРОВКИ ЛУЧЕЙ НА GPU

Тривиальная реализация трассировки лучей на GPU в виде одного монолитного ядра, как правило, показывает низкую производительность. Такая реализация может быть выполнена в виде простой процедуры, где одно ядро производит обход пространственной структуры данных, выполняет пересечение луча с примитивом, производит расчет цвета и генерацию новых лучей. В работе [10] проводится анализ эффективности различных реализаций трассировки лучей на GPU. Тривиальная реализация создаёт следующие трудности:

- 1) Большой объем использованной регистровой памяти;
- 2) Визуальный профилировщик указывает на узкое место в «инструкциях»;
- 3) Обращения в локальную память;
- 4) Профилировщик указывает на расходящиеся ветвления.

Эти проблемы тесно переплетены. Количество регистров напрямую влияет на занятость GPU. Занятость – это отношение количества потоков исполняющихся на GPU в данный момент к максимально возможному числу потоков. Занятость может служить как мерой производительности, так и индикатором узкого места.

Разделяемая память и регистры – это ограниченные по объему ресурсы. Аппаратный планировщик потоков может запустить лишь ограниченное число потоков одновременно. Предположим, что на потоковом мультипроцессоре всего X регистров, и компилятор выделил на каждый поток Y регистров. Тогда максимальное количество потоков, которое

может быть запущено, определяется как X / Y . В частности, на архитектуре NVIDIA Tesla 10 потоковый мультипроцессор располагает 16 * 1024 32-ух битными регистрами. Таким образом, если количество регистров на поток равно 32, то запустится не более 512 потоков.

Когда количество регистров превышает определенную планку, ожидаемым результатом становится низкая загрузка GPU. Дополнительно стоит помнить, что на GPU от количества потоков зависит и то, насколько хорошо будет покрываться латентность. Низкая занятость мультипроцессора приводит к тому, что обращения в память становятся узким местом.

Компилятор пытается уменьшить количество используемых регистров, сохраняя их при необходимости в локальную память и читая их содержимое оттуда. Локальная память имеет такую же латентность доступа, как и глобальная. Анализируя ассемблер CUDA PTX, можно увидеть такие повторяющиеся инструкции, как **local store** и **local load** в теле цикла. Это увеличивает нагрузку на шину памяти и, вдобавок к плохому покрытию латентности, приводит к ограничению производительности по памяти.

Наконец, когда группа потоков “warp” выполняет условный оператор и разные потоки из группы “warp” пошли разными путями, процессор вынужден будет выполнить обе ветви. Об этом свидетельствует наличие в профилировщике сообщений о расхождении потоков.

4.1 Предложенный конвейер трассировки лучей

Для упрощения процесса профилирования и отладки мы разбили конвейер на следующие этапы (рисунок 3):

- 1) **Ядро генерации лучей.** Это ядро, вычисляющее начало и направление первичных или вторичных лучей. Лучи упакованы в линейной памяти, как структура массивов.
- 2) **Проход луча по дереву.** В нашей реализации применяется Kd-дерево со стеком в локальной памяти. Так как нам известна заранее максимальная глубина дерева, мы можем консервативно оценить размер стека так, чтобы не превышать этой глубины. Стек хранит индекс на дальний узел и расстояние *tfar*. Это ядро сохраняет список непустых листьев Kd-дерева, который используется на следующем этапе. Для минимизации переключений между ядрами проход по дереву осуществляется до тех пор, пока не найдено некоторое количество непустых листьев. Список листьев записывается в заранее выделенный массив. Стек на локальной памяти теряется после переключения между вторым и третьим ядром, поэтому если пересечения не было найдено, мы можем продолжить обход дерева используя метод «restart» из [2].
- 3) **Пересечение луча с треугольниками.** Данное ядро перебирает по порядку сохраненные листья и тестирует луч на пересечение с треугольниками. Если пересечения внутри листа не было найдено, то начало луча модифицируется и отправляется обратно на этап обхода дерева. Если пересечение было найдено, луч отправляется дальше по конвейеру.
- 4) **Теневое ядро** вычисляет теневые лучи и осуществляет проверку видимости источников света.

- 5) На этапе **расчет освещения** вычисляется первичное освещение с затенением.
- 6) Цель **ядра материалов** создать вторичные лучи (например, отраженные или преломленные лучи). Эти лучи отправляются на этап обхода дерева.
- 7) Сохранение результата.

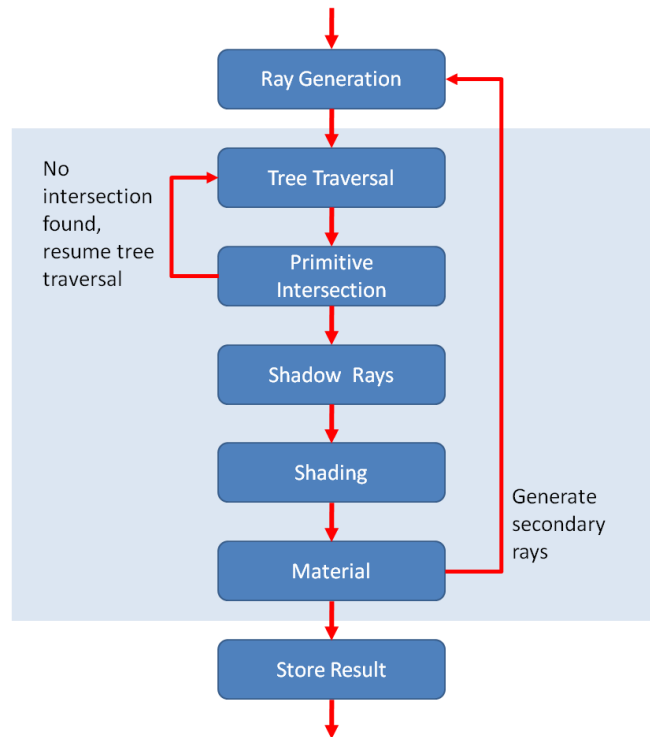


Рис. 3. Конвейер трассировки лучей

Такое разделение имеет следующие преимущества:

- 1) Анализ использования регистров: обход Kd-дерева уместается в 16 регистров, что дает максимальную занятость мультипроцессора. Пересечение луча с треугольником использует только 32 регистра, что дает 50% занятости на архитектуре Tesla 10. Расчет освещения оказался наиболее регистроемким участком, в виду особенностей использованной модели.
- 2) Уменьшение числа использованных регистров одновременно избавило код от существенной доли обращений в локальную память.
- 3) Разделение ядер сокращает количество расходящихся потоков. В виду того, что основные стадии выделены в отдельные ядра, единственной причиной для дивергенции осталась переменное число итераций в циклах (например, из-за переменного числа треугольников в листьях Kd-дерева)

Последнее – переменное количество выполняемой работы каждым потоком по-прежнему является проблемой. Например, при раздельных ядрах тестирование луча на пересечение с треугольниками не начнется до тех пор, пока не завершит работу ядро обхода дерева. Это не исключает ситуации, когда на исполнение ядра обхода для незначительного набора лучей (с наиболее долгим путем обхода дерева) тратится много времени. Чтобы уменьшить влияние таких предельных случаев, была реализована так

называемая модель «постоянных потоков» (persistent threads): каждый поток обрабатывает несколько лучей вместо одного. Весь экран разбивается на блоки, как показано на рисунке 4, и запускается такое количество блоков потоков, которое одновременно умещается на GPU. В такой реализации каждый блок потоков обрабатывает несколько блоков изображения. Это похоже на реализацию «множества лучей», описанную в работе [10], однако позволяет избежать необходимости использовать малоэффективные атомарные операции.



Рис. 4. Экран разделенный на блоки. Блоки одного цвета обрабатываются одним блоком потоков.

Наконец, получив хорошую производительность (в среднем ~ 50 М / сек лучей), мы объединили стадии обхода дерева и пересечения, в одно монолитное ядро.

Объединение этих двух этапов в одно ядро в сочетании с методом постоянных потоков так же уменьшает количество необходимых запусков ядер и упрощает управление потоками. Такая реализация в среднем на 5-10% быстрее и не требует дополнительных расходов по памяти.

5. ПРЕДЛОЖЕННЫЙ ПОДХОД

В большинстве пакетов фотореалистичной визуализации, для ускорения визуализации используют комбинацию распределенной трассировки лучей и кэша освещенности (irradiance cache). При таком подходе вторичная освещенность разделяется на 2 части: диффузную и отраженную. Отраженная часть рассчитывается с помощью распределенной трассировки лучей для каждого пиксела изображения. Диффузная часть рассчитывается лишь на некотором небольшом множестве точек в мировом пространстве, а во всех остальных точках интерполируется. Такой подход хорошо работает, если ДФО (Двулучевая Функция Отражения) материалов можно разделить на константную часть, которая одинакова для всех направлений и переменную, которая имеет максимумы в одном или нескольких направлениях.

Наш подход состоит из следующих шагов:

- 1) Выделить вторичную диффузную освещенность из интегрального уравнения.
- 2) Для расчета таких составляющих интегрального уравнения как: мягкие тени, четкие и нечеткие отражения, преломления,

дефокус камеры и размытие при движении, - использовать алгоритм трассировки путей (path tracing).

- 3) Для расчета вторичной диффузной освещенности использовать кэш освещенности.

Цель данного подхода заключается в том, чтобы с одной стороны увеличить когерентность лучей в алгоритме трассировки путей, выделив трассировку по всем направлениям в отдельный проход расчета освещения в точке кэша излучения, а с другой стороны – сократить количество трассируемых путей, введя выборки по значимости (importance sampling) и используя интерполяцию там, где это возможно.

Однако реализация этой идеи на текущих массивно-параллельных вычислительных системах требует специального подхода, поскольку остро встает проблема распределения работы.

На рис. 5 изображен пример сцены без гладкой (диффузной) вторичной освещенности. Черными кругами отмечены “простые” области – области, в которых для каждого пиксела достаточно пустить несколько лучей, чтобы интеграл освещенности сошелся. Красными квадратами выделены наиболее тяжелые для расчета участки изображения. В этих областях может потребоваться несколько сотен лучей на пиксел. При реализации трассировки путей на центральном процессоре это не создает дополнительных проблем, т.к. для каждого пиксела можно трассировать ровно столько лучей сколько нужно – последовательно, до тех пор пока для конкретного пиксела интеграл освещенности не сойдется.

На графических процессорах непредсказуемое распределение работы для пикселей осложняет ситуацию, так как заранее неизвестно, какой пиксел сколько лучей потребует. Было бы крайне неэффективно трассировать, например, 1000 лучей для каждого пиксела на экране только потому, что какой-то один из них требует 1000.

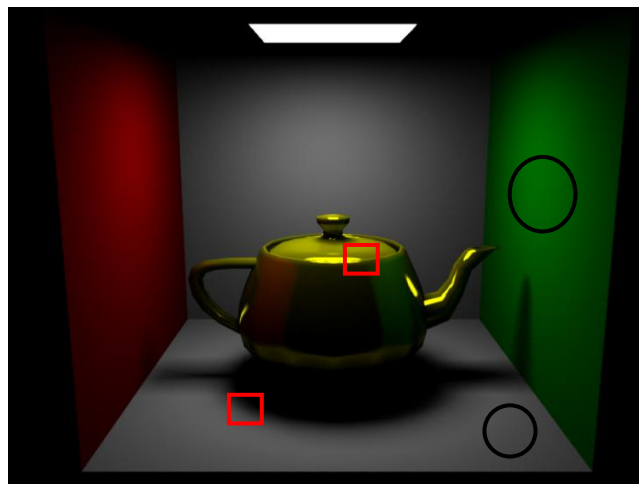


Рис. 5. Модель «чайник» внутри коробки Корнелла Cornell box (только первичное освещение)

5.1 Адаптивное распределение работы

Для того, чтобы сделать распределение работы адаптивным, мы разбили экран на блоки размером 8x8 или 16x16 пикселей. Внутри каждого блока мы нумеруем пиксели согласно кривой Мортон (рис. 6) для индексации всех данных, уникальных для каждого пиксела. Такая нумерация убирает

разрывы адреса при переходе к следующей строчке (в отличие от простой двумерной индексации), что позволяет более эффективно использовать важную особенность графических процессоров - объединение запросов к памяти (coalesced memory reads and writes).

В начале работы алгоритма все блоки добавляются в список активных блоков **active_list**. Пусть мы можем параллельно обрабатывать **TMAX** блоков. Это число зависит от количества доступной памяти на GPU. Пусть каждый блок имеет размер 16x16 пикселей.

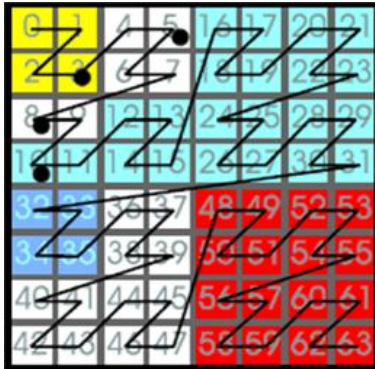


Рис. 6. Индексация пикселей используя кривую Мортон
Переменная **rays_per_pixel** отвечает за количество лучей, трассируемых на один пиксел.

```
var rays_per_pixel : integer :=1;

procedure Adaptive_Path_Tracing is
  active_list: list of Tile;
  active_array: array (0..TMAX-1) of Tile;
  sz,i: integer range 0..TMAX;
  tile : Tile;
begin
  subdivide screen to tiles;
  add all tiles to the active_list;

  while not active_list.empty():
    sz := min(active_list.size(), TMAX);
    active_array[0..sz] := active_list[0..sz];
    active_list := active_list[sz+1..end];

    Process_Tiles_On_GPU(active_array, \
                          sz, rays_per_pixel);

  for i in 0..TMAX-1:
    tile := active_array[i];
    if not tile.finished():
      active_list.push_back(tile);
  end for

  if active_list.size() < TMAX * 0.5:
    rays_per_pixel *= 2;
```

end while;

end Adaptive_Path_Tracing;

После одного шага трассировки путей (**Process_Tiles_On_GPU**), для каждого блока оценивается по некоторой метрике (мы вернемся к этому позже) можно ли считать его завершённым. Если блок завершён, он удаляется из списка. Если нет – добавляется обратно в список **active_list** и алгоритм повторяется с начала до тех пор, пока список **active_list** не пуст.

В процессе синтеза изображения некоторые блоки завершаются раньше, чем другие. Когда количество активных блоков меньше, чем **TMAX/2**, мы удваиваем количество работы, увеличивая тем самым в два раза количество трассируемых лучей на один пиксел. Таким образом, блоки, соответствующие наиболее сложным участкам изображения, будут долго оставаться активными, и все вычислительные ресурсы GPU в конечном счёте сосредоточатся на них.

```
procedure Process_Tiles_On_GPU (
  active_array array (0..TMAX-1) of Tile,
  sz : integer,
  rays_per_pixel : integer
) is
  tile_size : integer;
  rays_num : integer range 0..TMAX-1;
begin
  tile_size := 16*16;
  rays_num := tile_size*rays_per_pixel*sz;
  assert (rays_num <= TMAX);

  on the GPU:
    generate initial rays from the eye \
      according to the rays_per_pixel;
    trace exactly rays_num rays (paths in fact);
    sample result according to the rays_per_pixel;

end Process_Tiles_On_GPU;
```

При больших разрешениях (например, 1920x1200) наш подход позволяет экономить память и эффективно загрузить GPU работой.

Каждый блок представлен следующей структурой:

```
type Tile is record
  index      : integer;
  max_diff   : float;
  counter     : integer;
end record;
```

Массив таких структур **active_array** перемещается между центральным и графическим процессорами на каждом проходе алгоритма. Однако так как элементов массива в 256 раз меньше, чем количества потоков (блоки 16x16), такое копирование не является узким местом.

Поле 'index' содержит смещение для группы из 256 лучей в памяти GPU. Оно используется во время выборки нужных лучей и сохранения результата в соответствующее блоку место на изображении.

Для того, чтобы определить является ли блок завершённым, мы использовали следующий подход. Каждый пиксел накапливает частичные суммы интеграла освещённости в двух переменных **sum_{odd}** и **sum_{even}**, соответствующих суммам на всех чётных и нечётных шагах трассировки путей. На каждом втором шаге мы рассчитываем **max_diff** – значение, представляющее максимальную разницу (по всему блоку) между чётной и нечётной частичными суммами:

```
for i from 0 to 255 do:
    diff(i) := ||sumodd - sumeven||c
max_diff := max of all diff(i);
```

Таким образом, мы вычисляем 2 различные частичные суммы интеграла освещённости для каждого пиксела, и в тот момент, когда **max_diff** станет меньше чем некоторая допустимая ошибка, останавливаем процесс интегрирования для блока. То есть такой блок можно считать завершённым и удалить из списка активных блоков.

И наконец, поле **counter** представляет счетчик количества путей, которые уже были прослежены для данного блока (для всех пикселей внутри блока это количество одинаково).

5.2 Кэш освещённости

Наша реализация кэша освещённости аналогична реализации в работе [5]. Для каждого пиксела на GPU мы получаем позицию и нормаль. Затем на CPU в пространстве экрана строится квадродерево и между каждым текущим и дочерним узлом квадродерева вычисляются геометрические разрывы (разница в позиции и нормали). Когда геометрический разрыв становится меньше определенной величины, дальнейшее разбиение прекращается. Каждый узел квадродерева – это будущая точка кэша освещённости. Вдоль областей с сильными геометрическими разрывами мы добавляем случайным образом некоторое дополнительное число точек.

При расчете вторичного освещения из каждой точки полученного кэша мы испускаем лучи по всем направлениям и суммируем входящий свет с учётом диффузной природы материала. Чтобы уменьшить количество дивергентных групп потоков "warp" на GPU, мы собираем все лучи, выходящие из одной точки, в когерентные группы (лучи в такой группе слабо отличаются по направлениям) по $32 \cdot k$ лучей, где $k = 1, 2, 3$ и т.д.

После расчета вторичного освещения в точках кэша освещённости строится октодерево с множественными ссылками [11]. В процессе трассировки путей мы выбираем значения вторичной освещённости из кэша, используя алгоритм интерполяции, описанный в [12].

По-видимому, бесстековый алгоритм поиска в октодерево должен был бы быть эффективен на GPU. Однако наши эксперименты показали, что октодерево с множественными ссылками, описанное в [11], – не лучшее решение, потому что операция выборки значения из кэша занимает по времени примерно столько же, сколько занимает трассировка луча.

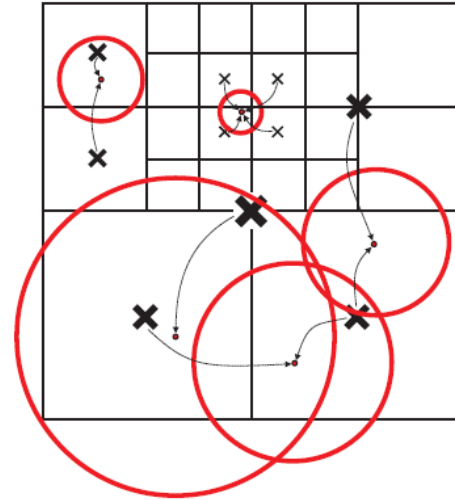


Рис. 7. Multiple reference octree as described in [11].

Основное преимущество октодерева с множественными ссылками в том, что алгоритм поиска ближайших точек в нем не рекурсивный, бесстековый. Для того чтобы найти все точки в заданном радиусе, нужно просто пройти от корня дерева до листа. Но цена за такой простой алгоритм высока – это множественные ссылки. В таком октодерево на каждую точку должны существовать ссылки из всех узлов октодерева, которые отстоят от неё на расстояние, равное радиусу поиска (рис. 7). При таком подходе количество ссылок может быть в 5-10 раз больше, чем количество самих точек. На CPU, где мы имеем большой L2 кэш, такая реализация, возможно, и приемлема. Однако на GPU косвенные обращения подобного характера снижают эффективность. Мы полагаем, что подход из работы [5] был бы более эффективным при реализации операции выборки из кэша на GPU, чем подход, описанный в [11]. Это одна из тем для будущих исследований.

5.3 Обзор решения в целом

Наша реализация на GPU состоит из множества маленьких CUDA ядер, каждое из которых выполняет одну узкую задачу. Мы сознательно отказались от архитектуры, предполагающей реализацию всего алгоритма в виде одного большого монолитного ядра с множеством состояний, по следующим причинам:

- 1) Монолитное ядро с множеством состояний ограничено по ресурсам наиболее тяжёлой своей частью. Например, если мы имеем сложный код, выполняющий расчет освещения, который занимает много регистров графического процессора, то этот код будет негативным образом влиять на производительность остальной части (например поиска в кд-дерево), так как ядро будет занимать столько ресурсов, сколько требует его наиболее ресурсоёмкая часть.
- 2) Некоторые пиксели могут потребовать довольно много лучей (больше тысячи), и все эти лучи в случае монолитного ядра будут трассироваться

последовательно до тех пор, пока интеграл не сойдется для всего блока. В нашей реализации для особо сложных участков мы трассируем для одного и того же пиксела несколько лучей параллельно, как только число лучей на пиксел становится больше одного.

- 3) Для отладки и профилирования сложного по своей природе кода, такого как трассировка лучей с множеством различных эффектов и материалов, наша архитектура с множеством маленьких ядер намного более удобна.

На каждом переотражении во время трассировки путей мы считаем прямое освещение, трассируя случайные теневые лучи, а не прямое диффузное освещение выбираем из кэша освещенности. Чтобы снизить относительную стоимость обращения в кэш освещенности, на сценах с мягкими тенями мы трассируем по 4 теневых луча при каждом переотражении во время трассировки пути.

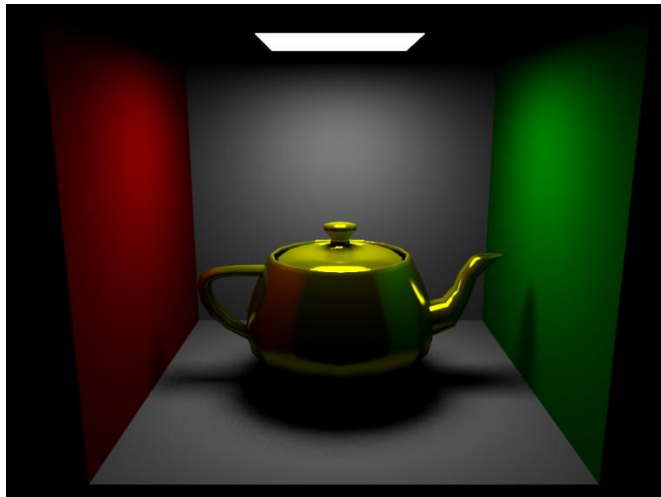


Рис. 8. VRay; Core 2 Quad, 6600; 62 сек в разрешении 1024x768

6. РЕЗУЛЬТАТЫ

Наш трассировщик лучей может обрабатывать 30-50 миллионов лучей в секунду на сцене 'Conference Room' и графическом процессоре GTX260. Это коррелирует с другими работами, посвященными ускорению трассировки лучей на GPU: [10], [13]. Мы использовали SAN кд-дерево для ускорения поиска пересечений.

Финальный рендер мы сравнили с промышленным пакетом VRay. Поскольку VRay – закрытый коммерческий продукт, задача попиксельного совпадения изображений не ставилась, поэтому рисунки 9 и 11 имеют заметные отличия.

На простых сценах наша реализация выигрывает у Vray в 4-5 раз. Однако наше решение хорошо масштабируется для массивных сцен и высоких разрешений, что можно увидеть на рис. 12, 13 и 14.

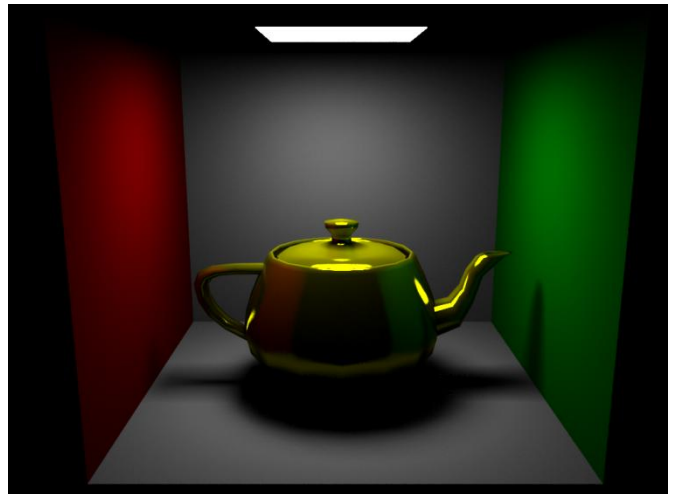


Рис. 10. Предложенная реализация; GTX260; 15 сек. в разрешении 1024x768



Рис. 9. VRay; Core 2 Quad; 6600; 153 сек. в разрешении 1024x768



Рис. 11. Предложенная реализация; GTX260; 31 сек. в разрешении 1024x768

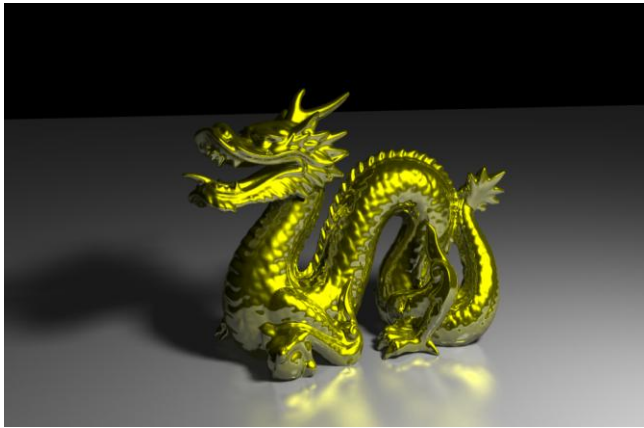


Рис. 12. 1920x1200. GTX260; 144 сек.

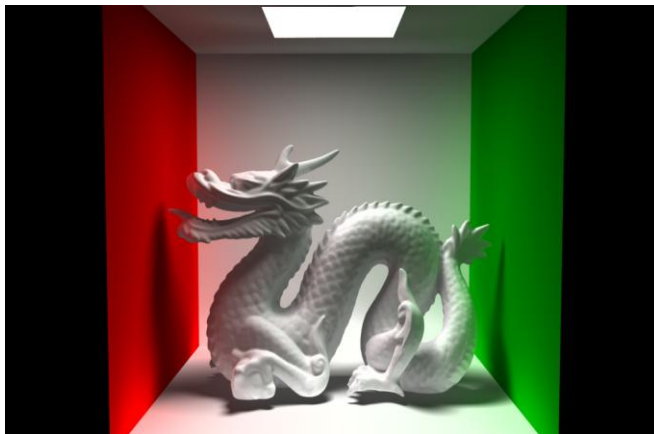


Рис. 13. 1920x1200. GTX260; 181 сек

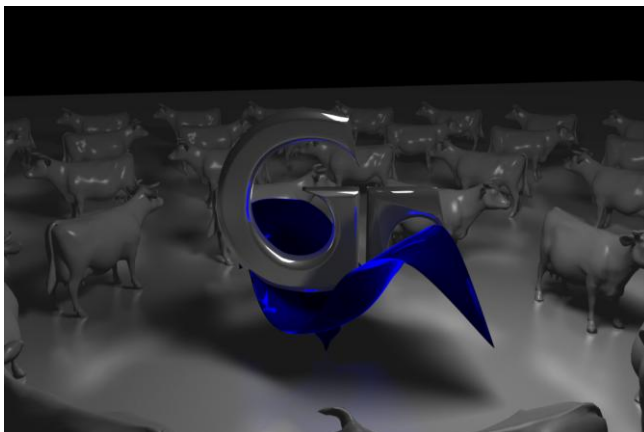


Рис. 14. 1920x1200. GTX260; 159 сек

7. ЛИТЕРАТУРА

- [1] Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. 2005. Ray tracing on programmable graphics hardware. In *ACM SIGGRAPH 2005 Courses* (Los Angeles, California, July 31 - August 04, 2005). J. Fujii, Ed. SIGGRAPH '05. ACM, New York, NY, 268. DOI=<http://doi.acm.org/10.1145/1198555.1198798>
- [2] Foley, T. and Sugerman, J. 2005. KD-tree acceleration structures for a GPU raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Los Angeles, California, July 30 - 31, 2005). HWS '05. ACM, New York, NY, 15-22. DOI=<http://doi.acm.org/10.1145/1071866.1071869>
- [3] Horn, D. R., Sugerman, J., Houston, M., and Hanrahan, P. 2007. Interactive k-d tree GPU raytracing. In *Proceedings of the 2007 Symposium on interactive 3D Graphics and Games* (Seattle, Washington, April 30 - May 02, 2007). I3D '07. ACM, New York, NY, 167-174. DOI=<http://doi.acm.org/10.1145/1230100.1230129>
- [4] THRANE N., SIMONSEN L. O.: A Comparison of Acceleration Structures for GPU Assisted Ray Tracing. Master's thesis, University of Aarhus, 2005. 2, 3
- [5] Wang R., Zhou K., Pan, M., and Bao, H. 2009. *An efficient GPU-based approach for interactive global illumination*. *ACM Trans. Graph.* 28, 3 (Jul. 2009), 1-8.
- [6] Fabianovski B., Dingliana J. *Interactive Global Photon Mapping*. In *Proceedings of the EUROGRAPHICS conference*, 2009. p. 1151-1159.
- [7] Frolov V., Ignatenko A. *Interactive GPU Ray Tracing and Photon Mapping*. In: *GraphiCon'2009.*; 2009. p. 255-262. (In Russian).
- [8] Jensen, H. W., Suykens F., Christensen Per H., Kato T. A *Practical Guide to Global Illumination using Photon Mapping*. SIGGRAPH 2002 Course Note #43. ACM, July 2002. (San Antonio, USA, July 21-26).
- [9] McGuire, M. and Luebke, D. 2009. Hardware-accelerated global illumination by image space photon mapping. In *Proceedings of the Conference on High Performance Graphics 2009* (New Orleans, Louisiana, August 01 - 03, 2009). S. N. Spencer, D. McAllister, M. Pharr, and I. Wald, Eds. HPG '09. ACM, New York, NY, 77-89. DOI=<http://doi.acm.org/10.1145/1572769.1572783>
- [10] Aila, T. and Laine, S. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009* (New Orleans, Louisiana, August 01 - 03, 2009). S. N.
- [11] Křivánek, J., Gautron, P., Ward, G., Jensen, H. W., Christensen, P. H., and Tabellion, E. 2008. *Practical global illumination with irradiance caching*. In *ACM SIGGRAPH 2008 Classes* (Los Angeles, California, August 11 - 15, 2008). SIGGRAPH '08. ACM, New York, NY, 1-20. DOI=<http://doi.acm.org/10.1145/1401132.1401213>
- [12] Pharr, M. and Humphreys, G. 2004 *Physically Based Rendering: from Theory to Implementation*. Morgan Kaufmann Publishers Inc.
- [13] Garanzha K., Loop C. *Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing*. In *Proceedings of the EUROGRAPHICS conference*, vol. 29 (2010), Number 2.