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

Кафедра информационно-измерительных систем

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Федотов Виктор Петрович

Реализация метода световых сеток на графическом ускорителе

Направление подготовки 230100.62 ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

Руководитель

Дебелов В. А.

д.т.н., в.н.с.

……

(подпись, дата)

Автор

Федотов В. П.

ФИТ, группа 8202

……

(подпись, дата)

Новосибирск, 2012 г.

Содержание

Введение ... 3
Глава 1. Обзор ... 5
1.1. Алгоритмы генерации теней ... 5
1.1.1. Обратная рекурсивная лучевая трассировка ... 6
1.1.2. Метод световых сеток ... 8
1.1.3. Оценка сложности ОРЛТ и МСС ... 9
1.2. Архитектура CUDA ... 10
Глава 2. Постановка задачи ... 13
Глава 3. Основные положения решения задачи ... 13
3.1. Требования к алгоритмам ... 13
3.2. Построение и обход сетки ... 13
3.3. Пересечение первичных лучей ... 14
3.4. Описание алгоритма для ОЛТ ... 15
3.5. Описание алгоритма для МСС ... 16
Глава 4. Реализация и численные эксперименты ... 19
4.2. Хранение сцены ... 19
4.4. Детали реализации МСС ... 20
4.4.1. Короткие тесты ... 20
4.4.2. Длинные тесты ... 22
4.5. Численные эксперименты ... 23
Заключение ... 26
Список литературы ... 27
Приложение А. Изображения тестовых сцен ... 28

ВВЕДЕНИЕ

Актуальность работы. Визуализация трехмерных сцен стала чрезвычайно популярной задачей в индустрии развлечений: кино, видеоигры, реклама. В связи с этим возросли требования к реалистичности рассчитываемых изображений, но вычислительные средства все еще недостаточно мощные, чтобы быстро проводить рендеринг сцен со сложными эффектами: отражения, преломления, дым и др. В частности это касается задачи генерации мягких теней.

В результате анализа, проведенного в диссертации Новикова И. Е. [1], алгоритмов генерации мягких теней было выявлено, что все они либо обладают хорошей производительностью, но страдают от графических артефактов, либо дают достаточно реалистичный результат, но требуют значительных вычислений. Однако метод световых сеток (МСС) [2], являясь модификацией обратной рекурсивной лучевой трассировки (ОРЛТ) [3], позволяет генерировать правдоподобные мягкие тени без артефактов быстрее сложных алгоритмов в случае сложных и сверхсложных сцен.

Генерация мягких теней в рамках рекурсивного ОРЛТ остается практически не исследованной на предмет ускорения расчетов, которое позволило бы получать качественное изображение за приемлемое время. В связи с этим было принято решение перенести часть вычислений с центрального процессора на графический процессор, используя возможности проведения расчетов общего назначения, и исследовать общее ускорение вычислений.

В качестве интерфейса для осуществления вычислений на GPU была выбрана программно-аппаратная архитектура CUDA [4], активно поддерживаемая компанией Nvidia. В последнее время CUDA бурно развивается, что приводит к значительному увеличению вычислительных мощностей видеокарт и позволяет проводить сложные расчеты даже на домашних компьютерах.

Компания Nvidia разработала и предоставила в общий доступ библиотеку для реализации рендеринга путем трассировки лучей под названием Optix [5]. Библиотека реализует трассировку лучей, поддерживает и ускорение за счет GPU, однако она не была использована в данной работе по причине высокий системных требований и отсутствия на рабочей станции подходящей видеокарты, требуется наличие видеокарты GeForce GTX 260 и выше.

Алгоритм, предложенный в данной работе, планируется реализовать в виде модулей к популярным системам визуализации, таких как Autodesk 3ds Max [6], Autodesk Maya [7] и др.

Алгоритм позволил бы значительно ускорить рендеринг сцен в видеофильмах, например, в учебных роликах, где не требуется точный расчет теней, а также получать предварительные кадры перед долгим расчетом финальной версии. При дальнейшем развитии и усовершенствовании (например, использовать алгоритм более экономного хранения световых точек, разработанного в работе Елагина [8]) мог бы стать полноценной альтернативой существующим методам.

Целью данной работы является разработка алгоритма для реализации метода световых сеток, используя программно-аппаратную архитектуру CUDA.

Достижение поставленной цели осуществляется за счет решения следующих задач:

1. Ознакомиться с алгоритмами ОРЛТ и МСС;
2. Ознакомиться с программно-аппаратной архитектурой CUDA;
3. Сформировать требования к программе тестирования и алгоритмам;
4. Реализовать программу тестирования;
5. Разработать и реализовать алгоритмы для ОЛТ и МСС в двух версиях: на CPU и GPU;
6. Провести эксперименты для тестовых сцен и сравнить времена для CPU и GPU;

Глава 1. Обзор

1.1. Алгоритмы генерации теней

Существуют разные математические модели освещенности сцены, на основе которых строится изображение трехмерной сцены. Например, для видеоигр производительность прорисовки сцены является наиболее важной характеристикой, поэтому качеством часто пренебрегают. В этой области применяется локальная модель освещенности, которая используется в OpenGL и DirectX. Если требуется высокое качество картинки, как в фильмах, то применяется глобальная модель освещенности, которая используется в таких трудоемких алгоритмах, как излучательность (radiosity), и методы Монте-Карло.

Но на практике чаще всего требуется алгоритм, сбалансированный по времени и качеству изображения. На современном этапе этому критерию наиболее полно отвечает модель освещенности Виттеда, используемая в ОРЛТ.

Стоит отметить, что под генерацией мягких теней понимается **расчет** мягкий теней и **имитация** мягких теней.

В модели Виттеда используются точечные источники света, которые могут дать только четкие тени, что не всегда соответствует тому, что мы наблюдаем в природе. Однако в рамках модели Виттеда можно получить мягкие тени, если заменить каждый точечный источник света множеством (сотни) точечных источников (см. рис. 1), находящихся рядом (данный метод реализован в программе Autodesk 3ds Max в виде модуля «Area shadows» [9] и используется по умолчанию). Это позволяет **рассчитывать** тени, получать правильный результат и качественную картинку, однако, так как время вычислений линейно зависит от количества источников света, то и время увеличивается в сотни раз.

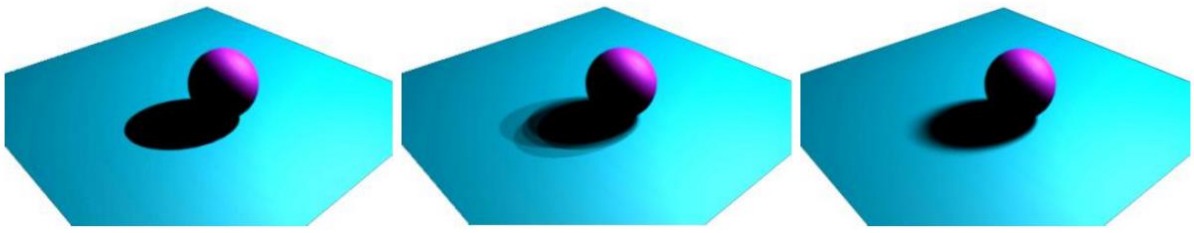


Рис. 1. Area shadows: 1 (слева), 5 (по центру), 225 (справа)

На практике часто требуется производить быстрый расчет, пусть даже с нарушениями в реалистичности и точности отображения теней, при этом **достаточно имитировать** мягкие тени (см. рис. 2).

Большинство решений для имитации мягких теней основаны на алгоритме теневых карт (АТК) [10] и алгоритме теневых объемов (АТО) [11]: насчитывается около 30 их модификаций.

Однако АТК, АТО и производные от них алгоритмы страдают от графических артефактов (см. рис. 2).

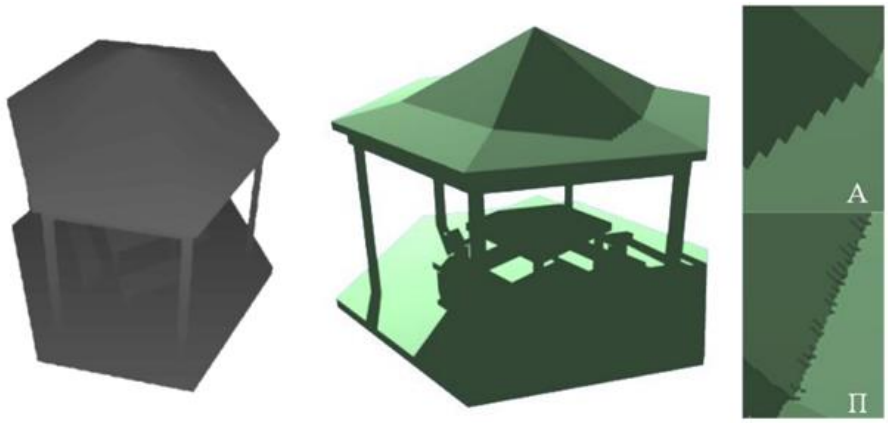


Рис. 2. Графические артефакты в результате использования АТК

А – алиасинг, П – протечки теней

Таким образом, многообразие алгоритмов генерации мягких теней подходит только для быстрого получения предварительных изображений.

1.1.1. Обратная рекурсивная лучевая трассировка

Рассмотрим задачу рендеринга – расчета реалистического изображения трехмерной сцены при помощи алгоритма ОРЛТ.

Пространственная сцена – это кусочно-непрерывная поверхность, представленная набором из $n0$ непрозрачных объектов или примитивов (треугольник, сфера, бокс и т.д. вплоть до фракталов), обладающих следующими свойствами:

В каждой точке поверхности определены свойства отражения, которые задаются коэффициентами: k_d – коэффициент диффузного отражения, k_s – коэффициент зеркального отражения.

Для поверхности определена операция пересечения с лучом.

В каждой точке поверхности определена нормаль.

Все объекты сцены задаются в декартовой мировой система координат.

Сцена освещается nL точечными источниками освещения L_i, специфицированными интенсивностями излучения I_i и позициями в пространстве LP_i, $i = 1 \dots nL$.

Камера — это набор параметров, характеризующих наблюдателя: позиция и ориентация в пространстве (где стоит наблюдатель, куда он смотрит, наклон вертикальной оси наблюдателя), вертикальный и горизонтальный углы поля зрения наблюдателя.

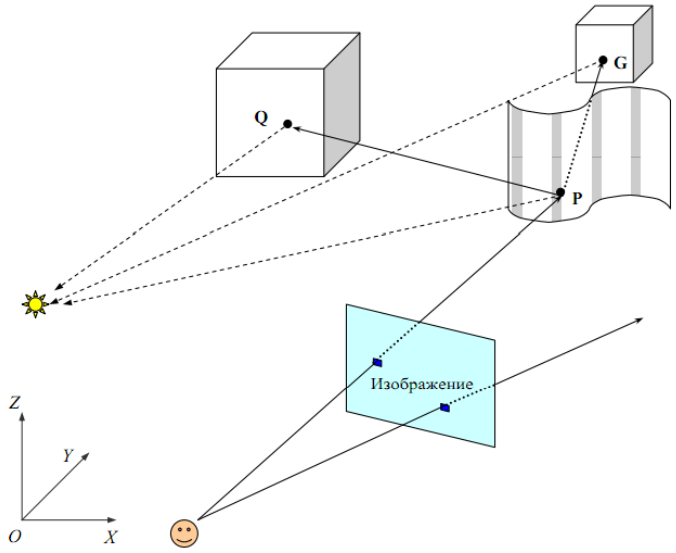


Рис. 3. Схема ОРЛТ

Алгоритма (см. рис. 3) заключается в испускании из камеры лучей, проходящих через центры пикселей экрана. Цветом пикселя является интенсивность в точке P – ближайшего пересечения луча со сценой. Для пикселя рассчитываются три значения интенсивности для каждого из цветовых каналов: красного, зеленого и синего, каждое значение вычисляется независимо от других. Интенсивность излучения в точке P вычисляется по формуле (1):

$$I(P) = I_a k_a + \sum_{i=1}^{nL} \left(V(LP_i, P) \cdot I_i \left[k_d\left(\vec{N}, \overrightarrow{D_i}\right) + k_s\left(\overrightarrow{R_i}, \vec{V}\right)^{pow} \right] \right) + k_s I_r \quad (1)$$

здесь $(,)$ -- скалярное произведение, все значения берутся в объектной точке P; I -- искомая интенсивность; I_a и k_a -- интенсивность и коэффициент отражения рассеянного света; pow –

степень зеркальности; $\vec{N}$, $\overrightarrow{D_l}$, $\overrightarrow{R_l}$ и $\vec{V}$ — нормаль, направление на i-ый источник, соответствующий ему вектор отражения и направление на камеру; I_r -- интенсивность, пришедшая с направления отраженного вектора. Функция видимости $V(LP_i, P)$ принимает два значения: 1, если точка P освещена i-ым источником света и 0 иначе.

Выражение без слагаемого $k_s I_r$ характеризует отсутствие рекурсии и отражений, т. е. в таком случае ОРЛТ заменятся нерекурсивной ОЛТ.

1.1.2. Метод световых сеток

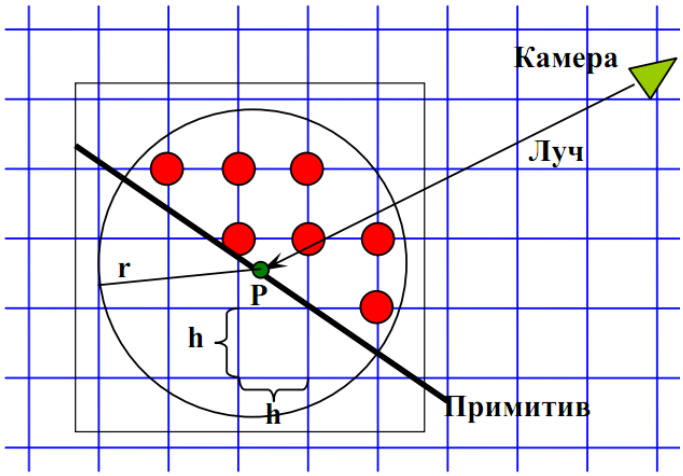


Рис. 4. Схема МСС

Для МСС является модификацией ОРЛТ и требует построение пространственной сетки из световых точек. Световые точки хранят в себе видимости до источников света, и эти значения вычисляются только при необходимости. В общем случае сетка неравномерная и плотность точек может варьироваться, однако работа с неравномерной сеткой проблематична и требует дополнительных вычислений, поэтому в нашем случае мы будем рассматривать только равномерную сетку с шагом h.

Видимость объектной точки P для i-го источника света вычисляется как взвешенная сумма значений видимости световых точек находящихся в интерполяционной полусфере $D_{LM}(P, r)$ (см. рис 4). Интерполяционная полусфера содержит в себе световые точки, находящиеся с положительной стороны нормали, на расстоянии не более r и в прямой видимости из точки P (2):

$$D_{LM}(P,r) = \{x_k : x_k \in LM, \|x_k - P\| < r, (n, x_k - P) > 0, V(P, x_k) = 1\} \quad (2)$$

Для МСС значение видимости в точке P вычисляется по формуле (3):

$$V(LP_i, P) = \chi(P, LP_i)\frac{1}{m}\sum_{k=1}^{m}[w(\|x_k - P\|) \cdot V(LP_i, x_k)] \quad (3)$$

здесь m — количество световых точек в $D_{LM}(P,r)$, функция $\chi(P, LP_i)$ может принимать значения: 1, если источник света освещает точку спереди и 0 если сзади.

В случае $m = 0$ используется формула (1).

1.1.3. Оценка сложности ОРЛТ и МСС

Время выполнения ОРЛТ значительно зависит от операций вычисления видимости объектных точек до источников света. Для выполнения этих операций необходимо проверить, пересекается ли отрезок (P, LP_i) со сценой, причем, длина этого отрезка сравнима с габаритами сцены, поэтому такой тест можно охарактеризовать как длинный тест. Пусть wh – разрешение изображения, nL – количество источников света и $cost_{long}$ – время выполнения длинного теста, тогда сложность ОРЛТ можно оценить как (4):

$$cost_{RT} = wh \cdot nL \cdot cost_{long} \quad (4)$$

В случае МСС производятся короткие тесты видимости световых точек из интерполяционной сферы. Так как количество световых точек, требующихся для вычисления видимостей всех объектных точек, меньше чем разрешение изображения, то и количество длинных тестов уменьшается по сравнению с ОРЛТ. Пусть $cost_{short}$ – время выполнения короткого теста, S_{sphere} – среднее количество световых точек в интерполяционной полусфере и S_{total} – общее количество задействованных световых точек на сцене. Тогда сложность МСС можно оценить как (5):

$$cost_{LM} = wh \cdot S_{sphere} \cdot cost_{short} + nL \cdot S_{total} \cdot cost_{long} \quad (5)$$

Видно, что всегда можно подобрать параметры wh и nL, такие что время расчета по МСС будет меньше чем по ОРЛТ.

Чтобы добиться приемлемого времени расчета по МСС необходимо, чтобы **время пересечения луча со сценой было приблизительно пропорционально его длине,** для этого используются ускоряющие структуры.

На практике часто используется Area Shadows, поэтому MCC, обладая преимуществом даже перед ОРЛТ, может генерировать визуально похожий результат при значительно меньшем количестве операций.

1.2. Архитектура CUDA

CUDA – программно-аппаратная архитектура, предоставляющая интерфейс для проведения вычислений общего назначения на графических ускорителях компании Nvidia. CUDA представляет собой расширение языка C, в котором предопределены следующие ключевые слова и типы:

Ключевые слова:

`__host__` – указывает компилятору, что функция, предваренная этим ключевым словом, исполняется только на центральном процессоре.

`__kernel__` – указывает компилятору, что эта функция (*ядро*) вызывается центральным процессором, а исполняется на GPU параллельно несколькими потоками.

`__device__` – указывает компилятору, что функция вызывается и исполняется только на GPU в контексте исполняющегося потока.

Типы: **`dim2`**, **`dim3`** (вектор из двух и трех целых), типы **`float2`**, **`float3`**, **`float4`**, **`int2`**, **`int3`**, **`int4`** и др.

Вызов ядра осуществляется следующим образом:

```
1. __global__ void kernel(/* params */);
2. int main() {
3.     kernel<<<grid, block>>>(/* params */);
4. }
```

Где `grid` и `block` имеют тип **`dim3`**.

Функции могут указываться одновременно как **`__host__`** и **`__device__`**, в таком случае компилятором будет создано два варианта кода – отдельно для CPU и отдельно для GPU.

Код, исполняющийся на GPU, имеет доступ только к видеопамяти, поэтому перед запуском вычислений следует подготовить память: выделить необходимый блок видеопамяти и скопировать туда нужные данные из оперативной памяти, для этого CUDA предоставляет

функции `cudaMalloc()`, `cudaMemcpy()` и другие. После вызовов ядер результат вычислений копируют из видеопамяти обратно в оперативную память.

Исполнение кода ядра на GPU ведется блоками потоков, которые организованы в виде двухмерной сетки (см. рис. 5)

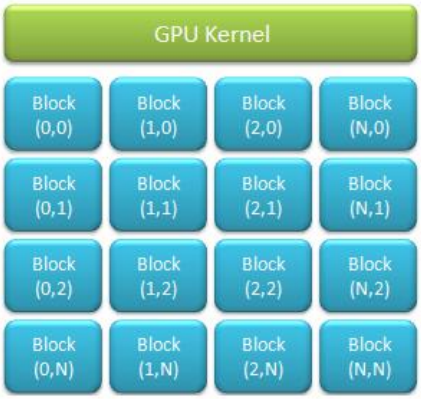


Рис. 5. Организация исполнения ядра в виде сетки блоков

Каждый блок состоит их потоков, которые являются непосредственными исполнителями вычислений. Потоки блока сформированы в виде трехмерной сетки (см. рис.6).

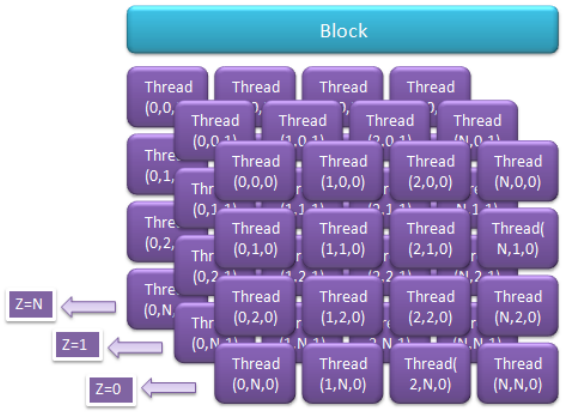


Рис. 6. Организация блока в виде трехмерной сетки потоков

Для идентификации номера потока используются специальные переменные `blockDim`, `blockIdx`, `threadIdx` типа ***dim3***, указывающие размер блока, координаты текущего блока и координаты потока внутри блока.

Блоки распределяются между мультипроцессорами CUDA (Streaming Multiprocessor, SM), выполнение инструкций внутри блока ведется по принципу SIMD варпами (warps) – группами из 32 потоков.

Существуют несколько типов видеопамяти:

- *Глобальная память*

Память с большим объемом (около 5,25 Гбайт на Nvidia Tesla C2070). Однако обращение к глобальной памяти требует сотни тактов, поэтому следует избегать лишних обращений к глобальной памяти.

- *Shared (разделяемая) память*

Быстрая память, объем которой ограничен от 16 Кбайт до 48 Кбайт на мультипроцессор. Доступ к shared памяти возможен только из потоков одного блока.

- *Локальная память*

Память, видимая только внутри потока, однако физически данные располагаются в глобальной памяти, поэтому следует избегать использования этого типа памяти.

- *Регистровая память*

Самая быстрая память, видимая только внутри потока. Регистры мультипроцессора распределяются между потоками.

- *Текстурная память*

Часть глобальной памяти, снабженная кэшем, оптимизированного для работы с одномерными и двухмерными массивами.

- *Константная память*

Память ограниченного размера с выделенным кэшем.

На мультипроцессоре ограниченное количество регистров и объем shared памяти, поэтому количество потоков, которые могут одновременно исполняться на мультипроцессоре, зависит от потребляемых ими ресурсов.

Глава 2. Постановка задачи

- Сцена задана в главе 1.1.1;
- Разработать программу визуализации сцен, в возможности которой входят:
  - Загрузка сцены;
  - Построение ускоряющей структуры для сцены;
  - Сохранение результата в виде BMP изображения;
- Разработать и реализовать алгоритмы ОЛТ и МСС в двух версиях:
  - На CPU;
  - На GPU (CUDA);
- Сравнить времена рендеринга на CPU и GPU для тестовых сцен:
  - Gazebo – 300 треугольников, 3 источника освещения;
  - Childpark – 27 000 треугольников, 2 источника освещения;
  - Garden – 234 000 треугольников, 1 источник освещения;

Глава 3. Основные положения решения задачи

3.1. Требования к алгоритмам

1. Использовать ускоряющую структуру – регулярную сетку;
2. Для версии на GPU использовать CUDA для ускорения для всех вычислений, которые можно распараллелить;
3. Экономно и эффективно использовать память для хранения сцены и временных данных;

3.2. Построение и обход сетки

В качестве ускоряющей структуры была выбрана регулярная сетка (Regular Grid) [12] в виду легкости её построения и обхода.

В процессе построения сетки, каждой ячейке назначается список индексов треугольников, попадающих в эту ячейку. Для этого нужно выполнить следующие шаги:

1. Для каждого треугольника вычислить габаритный бокс;
2. По габаритному боксу вычислить номера ячеек, в которые этот треугольник вероятно попадет;
3. Для каждой ячейки-кандидата запустить тест на пересечения треугольника и бокса.

Для обхода сетки используется алгоритм By [13] (см. рис. 7).

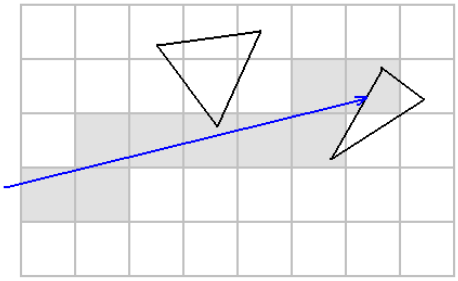


Рис. 7. Обход сетки по алгоритму By

В нашем случае не требуется ускорения построения сетки: уже построенную сетку можно хранить на диске и в случае необходимости оттуда загружать. Поэтому построение сетки производится на CPU, а затем просто копируется в видеопамять.

3.3. Пересечение первичных лучей

В CPU версии вычисления пересечений происходят обычным образом:

```
1.  __host__ void cast_primary_rays(...) {
2.      for (int y = 0; y < height; ++y) {
3.          for (int x = 0; x < width; ++x) {
4.              Ray ray = ray_for(x, y);
5.              intersect_primary_ray_and_save (ray);
6.          }
7.      }
8.  }
```

В GPU версии изображение разбивается на блоки, и каждому такому блоку соответствует блок потоков на GPU. В нашем случае размер блока составил – 16×16.

В таком случае псевдокод процедуры пересечения лучей:

```
1.  __host__ void cast_primary_rays(...) {
2.      dim3 block(16, 16);
3.      // Вычислить размеры сетки блоков (не сетки сцены!)
4.      dim3 grid = calc_grid_size(resolution, block);
5.
```

```
6.      gpu_cast_primary_rays<<<grid, block>>>(...);
7.  }
8.  __global__ void gpu_cast_primary_rays(...) {
9.      // Вычислить координаты пикселя
10.     int x = blockDim.x * blockIdx.x + threadIdx.x;
11.     int y = blockDim.y * blockIdx.y + threadIdx.y;
12.     if (x >= width || y >= height) exit;
13.     // По координатам пикселя вычислить луч
14.     Ray ray = ray_for(x, y);
15.     // Пересечь и сохранить результат
16.     intersect_primary_ray_and_save(ray, x, y);
17. }
```

3.4. Описание алгоритма для ОЛТ

Алгоритмы ОЛТ и МСС имеют общую часть – пересечение первичных лучей из камеры со сценой, поэтому было принято решение вынести эти действия в отдельную процедуру *cast_primary_rays()*.

Процедура *cast_primary_rays()* выполняет пересечение первичных лучей для каждого пикселя экрана и сохраняет координаты точек пересечения в памяти.

После того как вычислены все объектные точки, необходимо выполнить тесты видимости каждой объектной точки до источников освещения – этим занимается процедура *cast_shadow_rays(i)* для *i*-го источника освещения. Псевдокод ядра/процедуры:

```
1.  __global__ void cast_shadow_rays(int i) {
2.      int x = blockDim.x * blockIdx.x + threadIdx.x;
3.      int y = blockDim.y * blockIdx.y + threadIdx.y;
4.      if (x >= width || y >= height) exit;
5.      // Ничего не делать, если луч не столкнулся со сценой
6.      if (ray_missed(x, y)) exit;
7.      // Вычислить луч из пикселя (x, y) в направлении источника i
8.      Ray ray = for_from_to(x, y, i);
9.      // Пересечь и сохранить результат
10.     intersect_shadow_ray_and_save(ray, x, y, i);
11. }
```

Для CPU версии вычисления происходят аналогично `cast_primary_rays()` вложенными циклами.

Итого алгоритм принимает вид:

1. Вызов `cast_primary_rays()`.
2. Вызов `cast_shadow_rays(i)`, $i = 1 \dots nL$
3. Вычисление цветов пикселей по формуле (1).

В реализации на GPU процедуры `cast_primary_rays()` и `cast_shadow_rays(i)` определяются и вызываются как ядра.

3.5. Описание алгоритма для MCC

В версии для GPU изображение делится на блоки для выпуска первичных лучей. Внутри каждого блока (то есть пучка лучей) будет использоваться некоторое количество световых точек, в том числе световые точки, выходящие за границы блока. Это значит, что интерполяционные сферы соседних блоков будут пересекаться (см. рис. 8). Так как размер блока довольно мал, а шаг световой сетки и радиус интерполяционной сферы больше чем размер видимого пикселя, то интерполяционные сферы будут пересекаться значительно. Однако нам требуется, чтобы одни и те же световые точки не пересчитывались несколько раз.

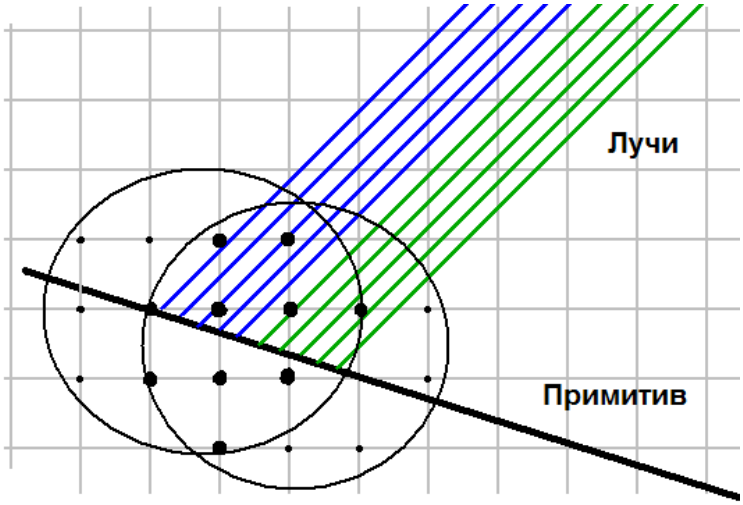


Рис. 8. Повторяющиеся световые точки соседних блоков.
Маленькие черные точки – задействованные световые точки

Большие – световые точки в пересечении интерполяционных множеств

Варианты организации вычислений световых точек:

Локальное вычисление световых точек: внутри каждого блока вычислять нужные световые точки и хранить часть световой сетки, используемую лучами блока, в shared памяти. Однако у этого варианта есть существенные недостатки:

- Перед тем, как вычислять световую точку нужно удостовериться, что она не была уже вычислена ранее, для этого потребуется доступ в глобальную память и семафор, это усложняет программирование.
- Необходимо оптимально хранить часть световой сетки в shared памяти. Лучи внутри блока могут проходить разное расстояние, перед тем, как столкнуться со сценой, это означает, что хранить сетку «цельным» параллелепипедом в памяти не получится, так как его размер заранее неизвестен (см. рис. 9).

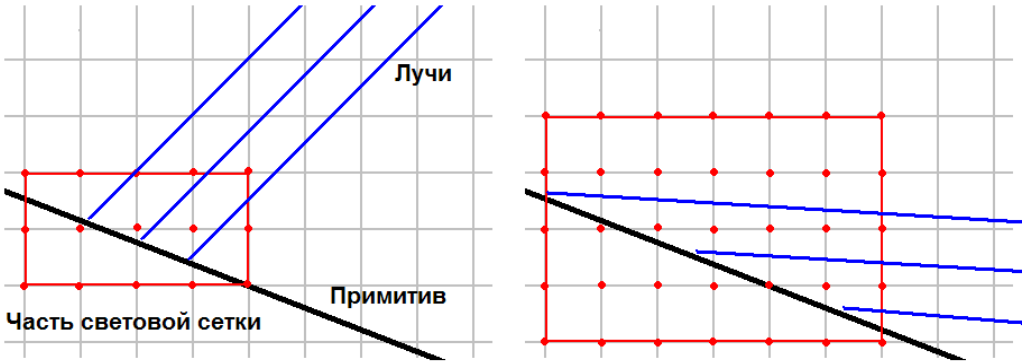


Рис. 9. Изменяющийся размер части световой сетки в случае хранения ее в виде параллелепипеда

- Если же хранить список задействованных световых точек каждого для луча в shared памяти, то встает вопрос о нехватке памяти при больших nL. Для блока размером 16×16 и радиуса интерполяционной сферы 3.1, количество световых точек на луч в шаблоне (глава) среднем 230, потребуется 7,25 Кбайт shared памяти для одного источника. Это много: максимальный объем shared памяти составляет от 16 Кбайт до 48 Кбайт на мультипроцессор (при $nL = 7$ на мультипроцессоре уже не хватает shared памяти даже для одного блока), а число работающих одновременно потоков напрямую зависит от количества используемых ресурсов, это значит, что при таком использовании памяти

снижается производительность. К тому же, в случае такой организации хранения, чтобы узнать о необходимости расчета световой точки необходимо пройтись циклом по интерполяционным сферам всех лучей внутри блока.

Глобальное вычисление световых точек: разбить алгоритм на последовательные шаги и не хранить часть световой сетки на GPU вовсе. Смысл таков:

1. Параллельно на GPU построить список световых точек интерполяционной сферы, нужных каждому лучу;
2. Одним потоком на CPU собрать список всех задействованных световых точек;
3. Параллельно на GPU вычислить видимости световых точек до источников освещения;
4. Одним потоком на CPU восстановить полную световую сетку.

Таким образом, разрешаются проблемы локального вычисления:

- Не нужно удостоверяться, что какие-либо световые точки не были вычислены – все они вычислены заранее;
- Не нужно хранить часть световой сети в shared памяти;
- Однопоточное составление списка задействованных световых точек и восстановление сетки не является узким местом (см. табл. 1);

Разрешение	h	r/h	Время
1024×1024	0,1	3,1	43.70 / **0.72**
2048×2048	0,1	3,1	171.76 / **2.8**

Таблица 1. Общее время / время на построение списка световых точек и восстановление сетки для сцены Garden на CPU (время в секундах).

Итого алгоритм принимает вид:

1. Вызвать **ядро** `cast_primary_rays();`
2. Вызвать **ядро** `short_tests();`

   Для каждой объектной точки на поверхности сцены строится список световых точек, входящих в интерполяционную сферу.
3. Скопировать результат работы `short_tests()` на CPU;
3. Исполнить процедуру `calculate_lm_points()` на CPU;

   Строит список всех задействованных на сцене световых точек.

4. Скопировать список световых точек в видеопамять.
5. Вызвать **ядро** `cast_lm_shadow_rays(i)`, $i = 1 \dots nL$

   Для каждой задействованной световой точки вычисляется видимость до i-го источника.
6. Скопировать результат работы `cast_lm_shadow_rays(i)` на CPU;
7. Вызвать **ядро** `create_lm()`.

   Создает полную световую сетку, хранящую в себе информацию о видимости всех световых точек до всех источников освещения.
8. Скопировать полную световую сетку в видеопамять.
9. Вызвать **ядро** `calc_pixels()`.

   Вычисляет цвета пикселей по формуле (2).
10. Скопировать результат `calc_pixels()` на CPU и сохранить на диске.

Псевдокод `cast_lm_shadow_rays()`:

```
1.  __global__ void cast_lm_shadow_rays(int i) {
2.      int j = blockDim.x * blockIdx.x + threadIdx.x;
3.      if (j >= points_num) exit;
4.      // Вычислить луч из j-ой световой точки списка в направлении
    источника i
5.      Ray ray = for_lm_from_to(j, i);
6.      // Пересечь и сохранить результат
7.      intersect_shadow_ray_and_save(ray, j, i);
8.  }
```

Глава 4. Реализация и численные эксперименты

4.1. Хранение сцены

Сетка хранится в памяти в “плоском” виде, в виде целочисленного массива `grid` в формате:

```
<grid>   ::= divx divy divz <offset>...<offset>
<offset> ::= -1 | cell_offset
<cell>   ::= -triangles_number triangles_offset | <grid>
```

div_x, div_y, div_z	– размерность сетки;
<offset>...<offset>	– последовательность из $div_x \times div_y \times div_z$ смещений узлов от начала массива;
<offset>	– смещение от начала массива, если -1 то пустая клетка;
<cell>	– формат ячейки;
triangles_offset	– начало списка вершин в массиве triangles;

triangles -- массив типа ***float4***, хранящий в себе данные о вершинах треугольников.

Три последовательно идущие вершины это соответственно: 1) первая вершина; 2) первая сторона треугольника; 3) вторая сторона треугольника;

Доступ к ним осуществляется через triangles[i*3], triangles[i*3+1] и triangles[i*3+2].

В случае GPU массив треугольников хранится в текстурной памяти для того, чтобы треугольники, находящиеся рядом с запрошенным треугольником, а значит находящиеся в той же ячейке, попадали в текстурный кэш.

4.2. Детали реализации MCC

Основные усилия по разработке MCC под GPU были направлены на экономное использование памяти.

4.2.1. Короткие тесты

Процедуре *short_tests*() требуется составить список задействованных световых точек (точнее их координат) для каждой объектной точки, однако, интерполяционная сфера каждого луча содержит в себе около 100 световых точек для $\frac{r}{h} = 3{,}1$.

Это значит, что если хранить список в виде 3 целых значений, то для изображения размером 1024×1024 потребуется 1024×1024×100×3×4 байт = 1,2 Гбайт – недопустимо много.

Для решения данной задачи было решено использовать шаблоны.

Шаблон – это список световых точек интерполяционной сферы, которые требуются для вычисления освещенностей объектных точек, попавших в одну клетку световой сетки (формула 6) (см. рис. 10):

$$DT_{LM}(r) = \bigcup_{P \in [0;1]^3} D_{LM}(P, r) \qquad (6)$$

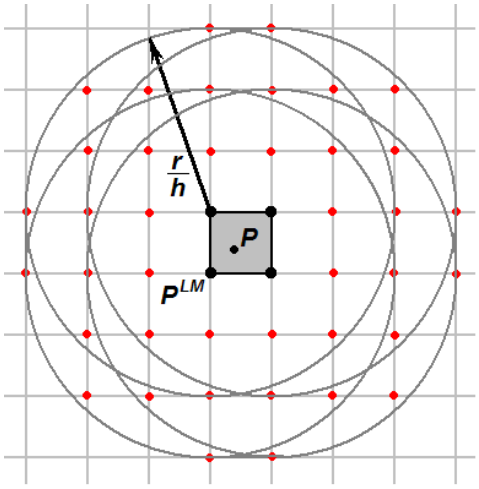


Рис. 10. Шаблон для $\frac{r}{h} = 3$

Шаблон используется следующим способом:

1. Для заданной объектной точки P вычисляется ближайшая световая точка P^{LM}, такая что, $P_x^{LM} \le P_x, P_y^{LM} \le P_y, P_z^{LM} \le P_z$
2. По шаблону $DT_{LM}(r)$ вычисляются точки, которые действительно попали в интерполяционную сферу:

$$D_{LM}(P, r) = \{x_k \in DT_{LM}(r) : \|x_k - P\| < r, (n, x_k - P) > 0, V(P, x_k) = 1\}$$

3. Заполняется битовая маска D^{used} подходящего размера (в нашем случае мы рассчитывали на 256 точек): k-ой световой точке x_k из $DT_{LM}(r)$ ставится в соответствие k-ый бит D_k^{used}:

$$D_k^{used} = \begin{cases} 1, & x_k \in D_{LM}(P, r) \\ 0, & \text{иначе} \end{cases}$$

Таким образом, для изображения 1024×1024 и радиуса интерполяционной сферы 3,1 будем хранить 256-битную маску для шаблона, для этого потребуется 1024×1024×32 байт = 32 Мбайт.

Псевдокод:

```
__global__ void short_tests() {
    int x = blockDim.x * blockIdx.x + threadIdx.x;
    int y = blockDim.y * blockIdx.y + threadIdx.y;
    if (x >= width || y >= height) exit;
    if (ray_missed(x, y)) exit;
    // Объектная точка первичного луча (x, y)
    float3 P = point(x, y);
    // Построить шаблон
    int3 template = build_template(P);
    // Изначально битовая маска нулевая
    bits256 mask = {0};
    for each lm_point in template:
        // Если световая точка действительно попадает в сферу
        if (length(lm_point - P) < r
         && dot(n, lm_point - P) > 0) {
            // И видима из точки P
            if (!intersect_segment_scene(lm_point, P))
            {
                setbit(mask, lm_point, P);
            }
        }
    }
}
```

4.2.2. Длинные тесты

Процедура строит список всех задействованных световых точек. Однако таких световых точек может быть много (см. табл. 2).

Сцена	h	r/h	S_{total}
Childpark	1	3,1	510 000
	0,5	3,1	1 574 000
Garden	0,1	3,1	102 000
	0,05	3,1	368 000

Таблица 2. Количество задействованных световых точек в зависимости от h и r/h

Поэтому было решено хранить в списке не 3 целых (координаты световой точки), а одно целое (7):

$$packed = div_x \cdot div_y \cdot z + div_x \cdot y + x \quad (7)$$

Таким образом, хоть максимальные размеры световой сетки и ограничены 32 битами целого числа, но достигается трехкратное уменьшение объема использованной памяти.

4.3. Численные эксперименты

Расчеты проводились на устройствах:

- CPU: Intel Core i7 950 @ 3.07 ГГц
- GPU: Nvidia Tesla C2070

На рис. 11 показан график зависимости времени расчета сцены Childpark на CPU в зависимости от количества источников освещения. Ускоряющая структура 160×120×140, разрешение 2048×2048, $h = 1, r/h = 2$.

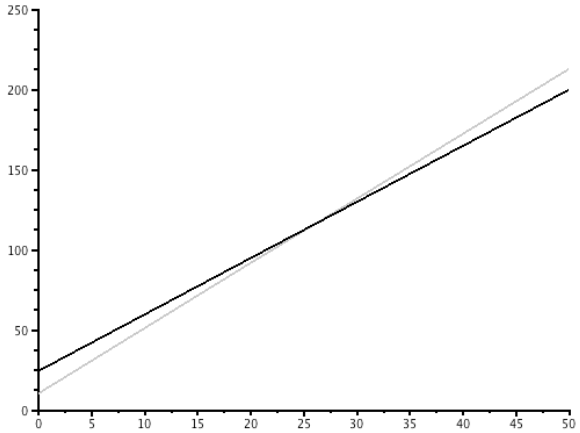


Рис. 11. Время в секундах в зависимости от количества источников освещения: серый – ОЛТ, черный - МСС

Качество мягких теней значительно зависит от шага сетки и радиуса интерполяционной сферы. На рис. 12-15 представлены результаты работы МСС с разными значениями r/h при $h = 1$:

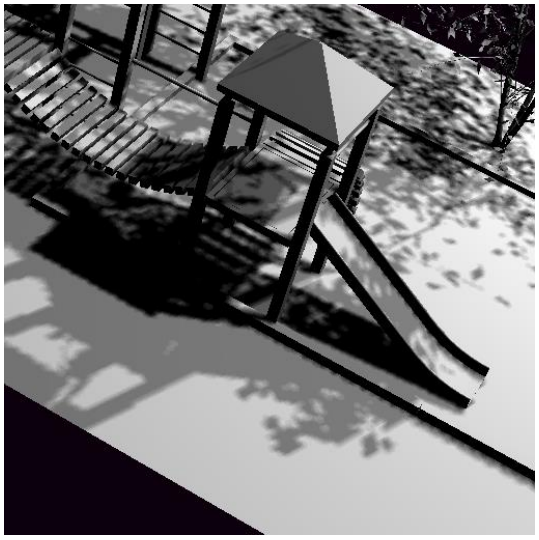


Рис. 12. $r/h = 1{,}5$

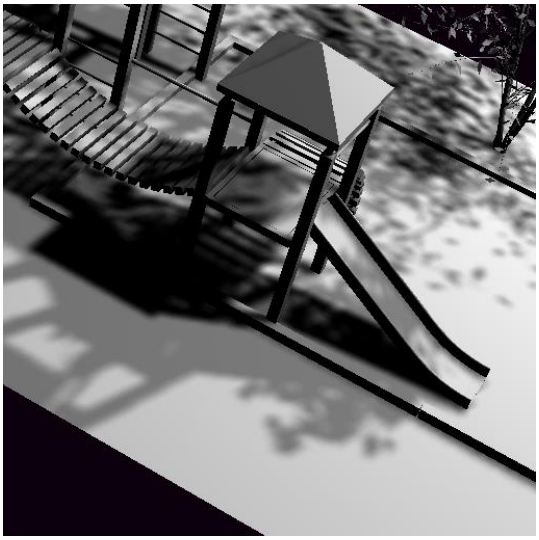


Рис. 13. $r/h = 2{,}5$

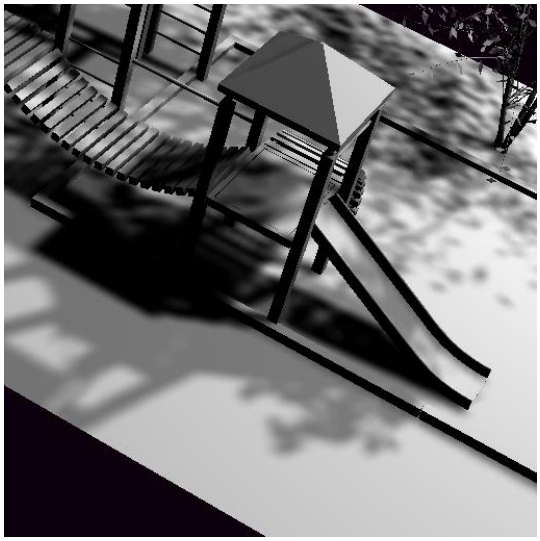


Рис. 14. $r/h = 3$

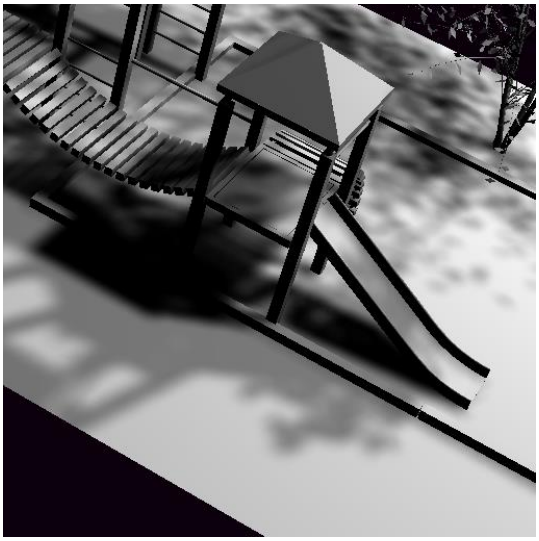


Рис. 15. $r/h = 4$

При значении радиуса интерполяционной сферы равным 3 получается удовлетворительный результат, поэтому в экспериментах будем использовать 3,1.

Результаты вычислений на сцене “Gazebo”, ускоряющая структура 15×10×20, 3 источника:

Разрешение	h	r/h	Время на CPU	Время на GPU
1024×1024	2	2,1	8,46	0,75
		3,2	23,06	1,77
	1	2,1	9,27	0,94
		3,2	24,28	1,95
2048×2048	2	2,1	33,05	2,87
		3,2	91,06	6,81
	1	2,1	35,07	3,12
		3,1	94,41	7,09

Результаты вычислений на сцене “Childpark”, ускоряющая структура 160×120×140, 2 источника:

Разрешение	h	r/h	Время на CPU	Время на GPU
1024×1024	1	2,1	9,94	0,81
		3,1	22,65	1,66
	0,5	2,1	12,97	1,8
		3,1	26,57	2,69
2048×2048	1	2,1	34,93	2,52
		3,1	82,71	5,55
	0,5	2,1	37,38	3,54
		3,1	83,03	6,59

Результаты вычислений на сцене “Garden”, ускоряющая структура 200×60×280, 1 источник:

Разрешение	h	r/h	Время на CPU	Время на GPU
1024 × 1024	0,1	2,1	8,8	0,63
		3,1	25,08	1,59
	0,05	2,1	7,63	0,64
		3,1	19,48	1,45
2048 × 2048	0,1	2,1	34,2	2,30
		3,1	99,05	5,69
	0,05	2,1	28,55	2,23
		3,1	75,29	5,16

Заключение

В результате дипломной работы:

1. Был разработан и реализован алгоритм для MCC на CPU и GPU.
2. Была разработана тестовая программа для проведения экспериментов.
3. Было показано, что алгоритм MCC успешно распараллеливается, и на некоторых сценах дает ускорение в 17 раз.
4. Также в результате учета особенностей CUDA, а именно, ограниченного объема видеопамяти, был разработан новый элемент в алгоритме – шаблон памяти.

Предложенный алгоритм можно использовать не только на специальных серверах, но и на домашних компьютерах.

При дальнейшем развитии MCC и реализации метода на GPU возможно его полноценное использование в видеографике в качестве полноценной альтернативы существующим методам.

Литература

[1] Дебелов В. А., Новиков И. Е. Разработка алгоритмов ускорения расчета изображений трехмерных сцен по методу световых сеток.

[2] Дебелов В. А., Васильева Л. Ф., Новиков И. Е. Развитие метода световых сеток для алгоритма лучевой трассировки: аппроксимация решения, реализация на графическом акселераторе // Тр. 15-й междунар. конф. по компьютерной графике и ее приложениям Графикон'2005. – Новосибирск, 2005. – С. 355–359.

[3] Whitted T. An Improved Illumination Model for Shaded Display / Commun. ACM. – 1980. – Vol. 23, № 6. – P. 343-349.

[4] Параллельное программирование и вычислительная платформа | CUDA | NVIDIA. URL: http://www.nvidia.ru/object/cuda_home_new_ru.html (дата обращения: 06.06.2012)

[5] Программный ускоряющий движок NVIDIA® OptiX. URL: http://www.nvidia.ru/object/optix_ru.html (дата обращения: 06.06.2012)

[6] 3ds Max – 3D-моделирование, анимация и рендеринг – Autodesk. URL: http://www.autodesk.ru/adsk/servlet/pc/index?siteID=871736&id=14642369 (дата обращения: 06.06.2012)

[7] Maya – Продукт для 3D-анимации – Autodesk. URL: http://www.autodesk.ru/adsk/servlet/pc/index?siteID=871736&id=14657576 (дата обращения: 06.06.2012)

[8] Елагин В. А. Адаптация метода световых сеток для генерации мягких теней в 3ds Max.

[9] Paul S. Heckbert and Michael Herf. Simulating soft shadows with graphics hardware. Technical Report CMU-CS-97-104, Carnegie Mellon University, 1997.

[10] Williams L., Casting curved shadows on curved surfaces // Computer Graphics. – 1978. – Vol. 10, № 2. – P. 270-274.

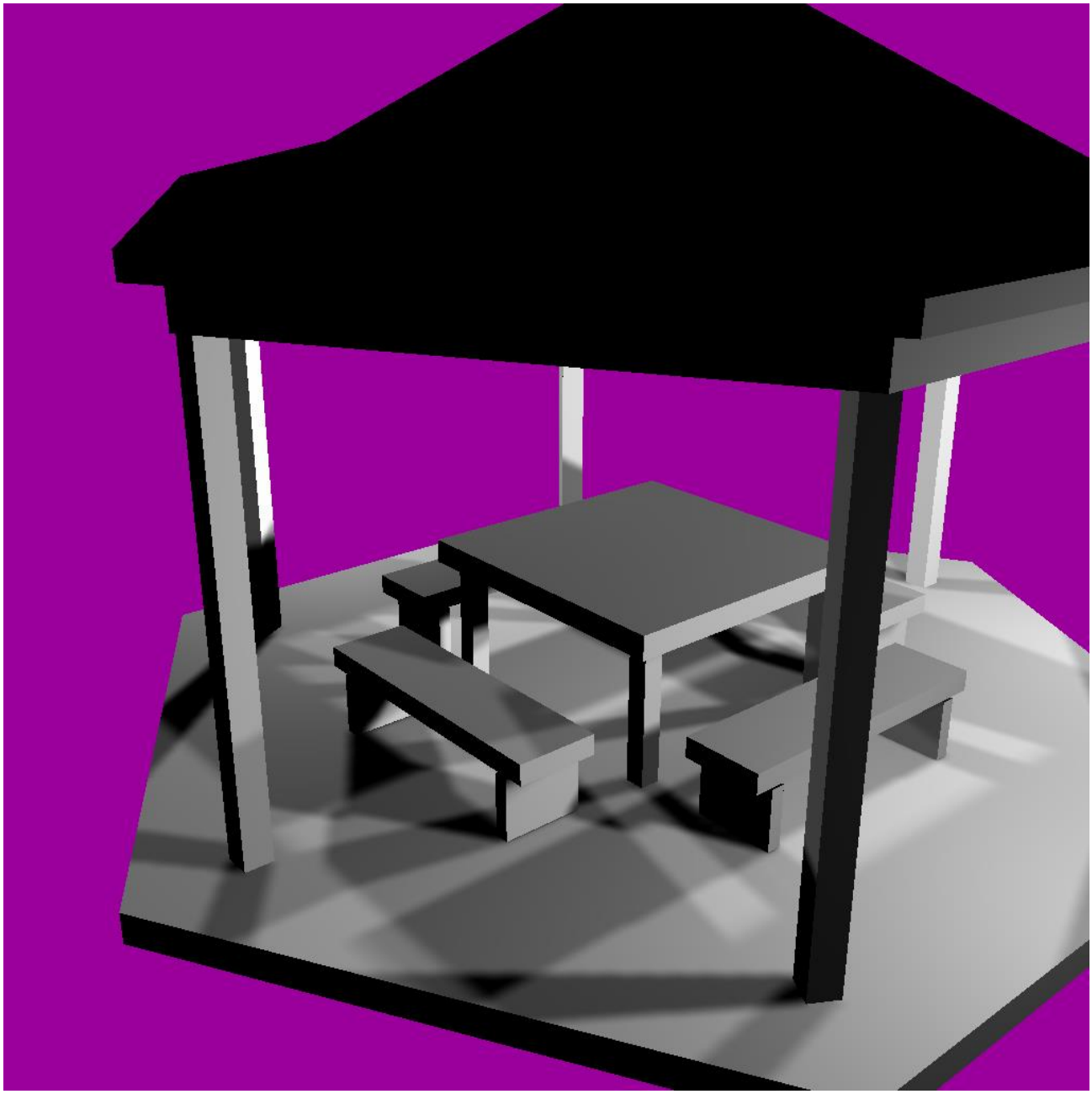
[11] Crow F. Shadow Algorithms for Computer Graphics // Computer Graphics. – 1977. – Vol. 11, № 2. – P. 242-247.

[12] A. Fujimoto and K. Iwata, “Accelerated Ray Tracing,” Proc. CG Tokyo ’85, pp. 41-65.

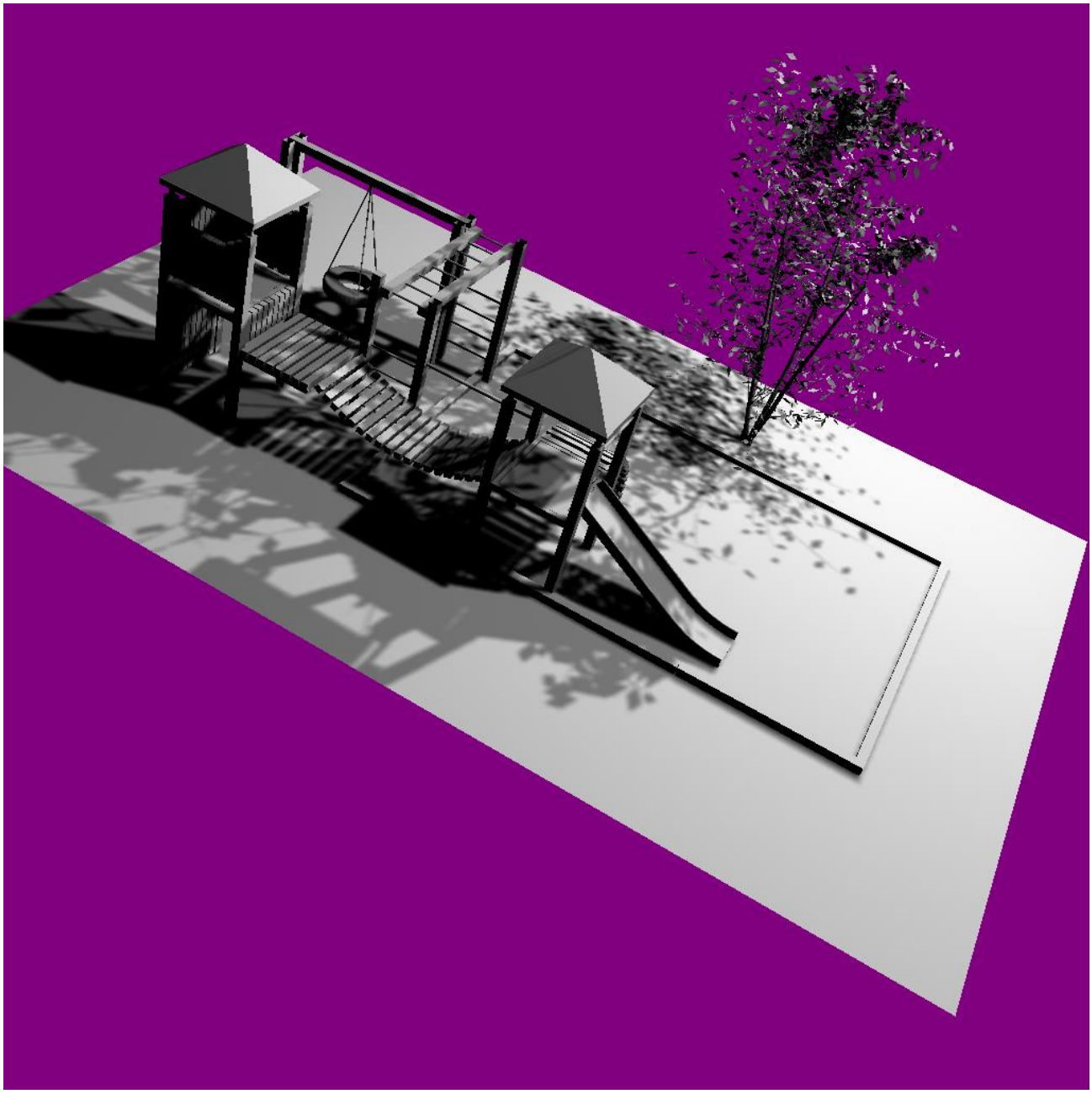
[13] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In Proc. Eurographics ’87, pages 3–10. Elsevier Science Publishers, Amsterdam, North-Holland, August 1987.

Приложение А
(рекомендуемое)

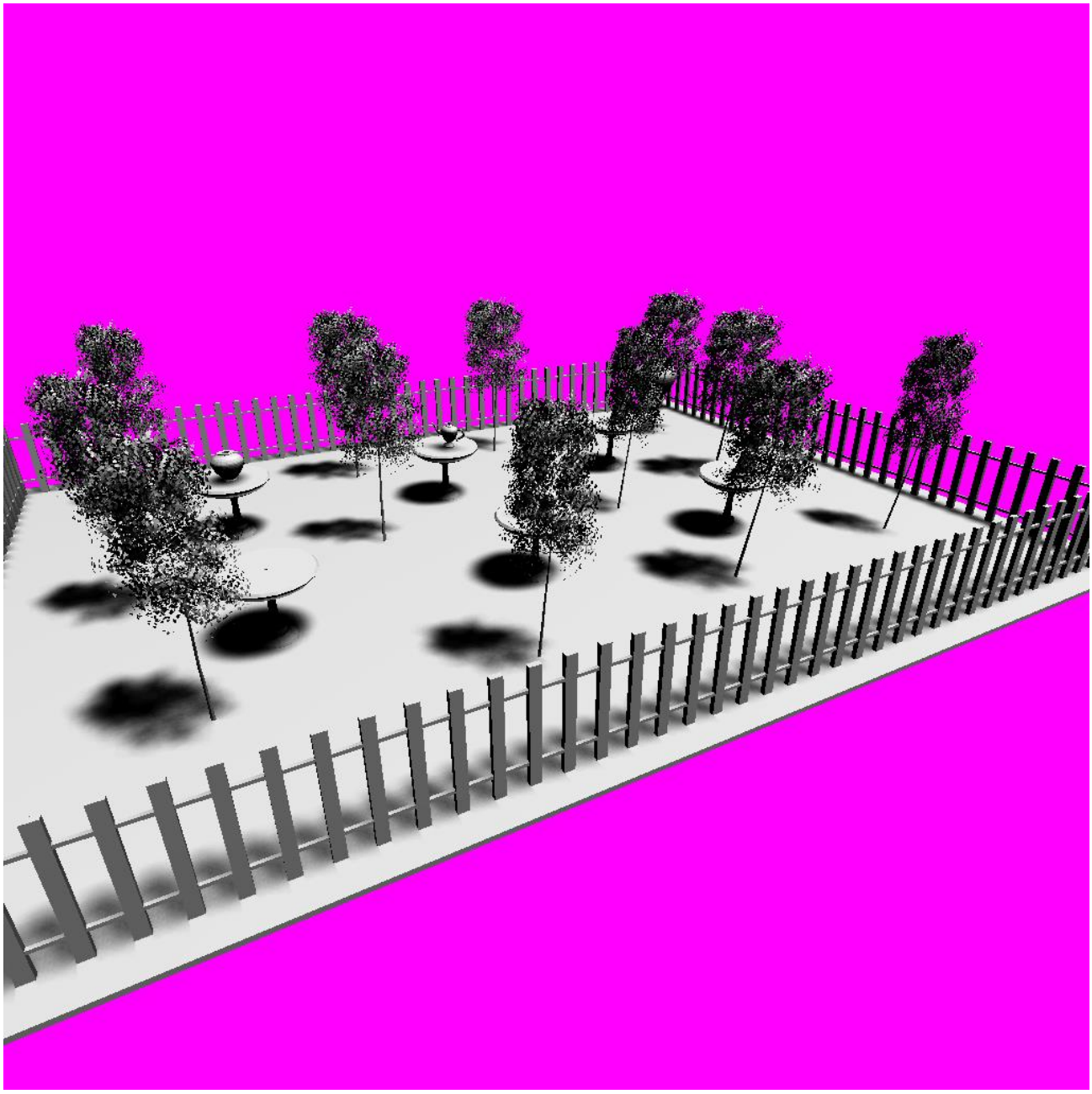
Изображения тестовых сцен



Сцена «Gazebo», 300 треугольников, 3 источника освещения $h = 1, r/h = 3{,}1$



Сцена «Childpark», 27 000 треугольников, 2 источника освещения $h = 0{,}5, r/h = 3{,}1$



Сцена «Garden», 234 000 треугольников, 3 источника освещения $h = 0{,}05, r/h = 3{,}1$