**Содержание**

Введение……………………………………………………………………...2

1.Анализ предметной области……………………………………………....3

1.1 Основные понятия метода трассировки лучей………………….3

1.2 Основные понятия архитектуры CUDA………………………....4

2.Описание проекта……………………………………………………….....5

2.1 Использование CUDA для распараллеливания алгоритма трассировки лучей…………………………………………………………...5

Заключение…………………………………………………………………...9

Список использованной литературы……………………………………....10

**Введение**

На данный момент технология трассировки лучей очень популярно во многих сферах, связанных с графикой. Особенно часто данная технология используется сейчас в игровой индустрии. При помощи данной технологии появилась возможность сделать любые объекты очень реалистичными благодаря обработке отражения и преломления от объктов.

Однако, несмотря на все преимущества, данная технология требует больших вычислительных мощностей. Далее мы рассмотрим различные способы оптимизации при реализации трассировки лучей, в особенности – распараллеливание при помощи видеокарты.

1. **Анализ предметной области**
   1. **Основные понятия метода трассировки лучей**.

**Трассировка лучей** (англ[.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Ray tracing*) — 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). Сегодня данная технология имеет очень большое влияние на игровую индустрию, так как позволяет получить намного более реалистичное изображение.

**Достоинства и недостатки трассировки лучей:**

**Достоинства:**

* возможность рендеринга гладких объектов без аппроксимации их полигональными поверхностями (например, треугольниками);
* вычислительная сложность метода слабо зависит от сложности сцены;
* 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;
* отсечение невидимых поверхностей, перспектива и корректное изменения поля зрения являются логическим следствием алгоритма.

**Недостатки:**

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](https://ru.wikipedia.org/wiki/%D0%A1%D0%B3%D0%BB%D0%B0%D0%B6%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) в определённых местах модели. Также это регулирует отражение лучей и эффекты преломления, и в целом — степень фотореалистичности изображения.

Методы оптимизации метода трассировки лучей.

На данный момент существует много различных оптимизаций для данного метода, например:

* Распараллеливание алгоритма
* Использование k-мерных деревьев для нахождения пересечения с объектом без линейного поиска по всем объектам.
* Решение интегрального уравнения рендеринга.

**1.2** **Основные понятия архитектуры CUDA**.

**CUDA**  — программно-аппаратная архитектура [параллельных вычислений](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F), которая позволяет существенно увеличить вычислительную производительность благодаря использованию [графических процессоров](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) фирмы [Nvidia](https://ru.wikipedia.org/wiki/Nvidia).

Сама технология CUDA (компилятор nvcc.exe) вводит ряд дополнительных расширений для языка C++, которые необходимы для написания кода для GPU:

1. Спецификаторы функций, которые показывают, как и откуда будут выполняться функции.
2. Спецификаторы переменных, которые служат для указания типа используемой памяти GPU.
3. Спецификаторы запуска ядра GPU.
4. Встроенные переменные для идентификации нитей, блоков и других параметров при использовании кода в ядре GPU.
5. Дополнительные типы переменных.

Как было сказано, спецификаторы функций определяют, как и откуда буду вызываться функции. Всего в CUDA 3 таких спецификатора:

* **\_\_host\_\_** — выполнятся на CPU, вызывается с CPU (в принципе его можно и не указывать).
* **\_\_global\_\_** — выполняется на GPU, вызывается с CPU.
* **\_\_device\_\_** — выполняется на GPU, вызывается с GPU.

Спецификаторы запуска ядра служат для описания количества блоков, нитей и памяти, которые вы хотите выделить при расчете на GPU. Синтаксис запуска ядра имеет следующий вид: myKernelFunc<<<gridSize, blockSize, sharedMemSize, cudaStream>>>(float\* param1,float\* param2), где

* **gridSize** – размерность сетки блоков (dim3), выделенную для расчетов.
* **blockSize** – размер блока (dim3), выделенного для расчетов.
* **sharedMemSize** – размер дополнительной памяти, выделяемой при запуске ядра.
* **cudaStream** – переменная cudaStream\_t, задающая поток, в котором будет произведен вызов.

myKernelFunc – функция ядра (спецификатор \_\_global\_\_). Некоторые переменные при вызове ядра можно опускать, например sharedMemSize и cudaStream.

Также стоит упомянуть о встроенных переменных:

* **gridDim** – размерность грида, имеет тип dim3. Позволяет узнать размер гридa, выделенного при текущем вызове ядра.
* **blockDim** – размерность блока, так же имеет тип dim3. Позволяет узнать размер блока, выделенного при текущем вызове ядра.
* **blockIdx** – индекс текущего блока в вычислении на GPU, имеет тип uint3.
* **threadIdx** – индекс текущей нити в вычислении на GPU, имеет тип uint3.

**2. Описание проекта.**

**2.1** **Использование CUDA для распараллеливания алгоритма трассировки лучей.**

В самой программе необходимо выполнить следующие этапы:

* Получение данных для расчетов.
* Скопировать эти данные в GPU память.
* Произвести вычисление в GPU через функцию ядра.
* Скопировать вычисленные данные из GPU памяти в ОЗУ.
* Высвободить используемые ресурсы.

Следующий код выделяет память под все классы для работы:

int num\_pixels = x \* y;

size\_t fb\_size = num\_pixels \* sizeof(vec3);

vec3 \*fb;

hitable\_list \*\*d\_world;

camera \*\*d\_camera;

hitable \*\*d\_list;

curandState \*d\_rand\_state;

checkCudaErrors(cudaMallocManaged(&fb, fb\_size));

checkCudaErrors(cudaMalloc(&d\_world, sizeof(hitable\_list \*)));

checkCudaErrors(cudaMalloc(&d\_camera, sizeof(camera\*))); checkCudaErrors(cudaMalloc(&d\_rand\_state, num\_pixels \* sizeof(curandState)));

create\_world <<< 1, 1 >>> (d\_list, d\_world, d\_camera);

checkCudaErrors(cudaGetLastError());

checkCudaErrors(cudaDeviceSynchronize());

Макрос checkCudaErrors представляет функцию для обработки любых исключений, появившихся по ходу работы программы

Метод create\_world, который запускается на GPU, создает различные объекты мира. Он принимает параметрами указатели на выделенную память под объекты, список объектов и начальное положение камеры (точки для испускания луча).

Далее мы инициализируем размер блока и запускаем отдельное ядро для вычисления rand\_state. Рандом используется для получения случайного угла отражения от поверхности:

\_\_device\_\_ vec3 color(ray& r, hitable\_list \*\*world, curandState \*local\_rand\_state) {

ray cur\_ray = r;

float cur\_attenuation = 1.0;

vec3 tmp = random\_direction(local\_rand\_state);

for (int i = 0; i < 25; i++) { // this is to find the degree of darkening

hit\_record rec;

if ((\*world)->hit(cur\_ray, 0.0001, FLT\_MAX, rec)) {

cur\_attenuation \*= 0.5;

cur\_ray = ray(rec.p, rec.normal + tmp);

}

else {

vec3 unit\_direction = unit\_vector(cur\_ray.direction());

float t = 0.5 \* (unit\_direction.x() + 1.0);

vec3 c = (1.0 - t) \* vec3(1.0, 1.0, 1.0) + t \* vec3(0.5, 0.7, 1.0);

return cur\_attenuation \* c;

}

}

return vec3(0.0, 0.0, 0.0);

}

Метод color итерационно проверяет пересечения луча с объектом. В случае пересечения мы запускаем отраженный луч от точки пересечения. Изначально луч имеет направление нормали к точке, но при помощи метода random\_direction мы случайным образом генерируем смещение для отраженного луча. Если же пересечения не было найдено, считаем, что мы попали на излучающий свет объект и возвращаем измененный при помощи коэффициента cur\_attenuation цвет. Метод hit класса hitable\_list представляет собой линейный поиск пересечения по всем объектам (в данном случае – сферам). При большой загруженности схемы данная реализация не является оптимальной. Именно для этого используются k-d деревья, позволяющие найти пересечение за логарифмическую сложность.

Метод для нахождения точки пересечения луча со сферой представлен ниже:

\_\_device\_\_ bool hit(const ray& r, float t\_min, float t\_max, hit\_record & rec)const {

vec3 oc = r.origin() - center;

float a = dot(r.direction(), r.direction());

float b = dot(oc, r.direction());

float c = dot(oc, oc) - radius \* radius;

float d = b \* b - a \* c;

if (d > 0) {

float temp = (-b - sqrt(d)) / a;

if (temp < t\_max && temp > t\_min) {

rec.t = temp;

rec.p = r.point\_at\_parameter(rec.t);

rec.normal = (rec.p - center) / radius;

return true;

}

temp = (-b + sqrt(d)) / a;

if (temp < t\_max && temp > t\_min) {

rec.t = temp;

rec.p = r.point\_at\_parameter(rec.t);

rec.normal = (rec.p - center) / radius;

return true;

}

}

return false;

}

Рассмотрим метод render, который вызывает метод color:

\_\_global\_\_ void render(vec3 \*fb, int max\_x, int max\_y, int k, camera \*\*cam, hitable\_list \*\*world, curandState \*rand\_state) {

int i = threadIdx.x + blockIdx.x \* blockDim.x;

int j = threadIdx.y + blockIdx.y \* blockDim.y;

if ((i >= max\_x) || (j >= max\_y))

return;

int pixel\_index = j \* max\_x + i;

curandState local\_rand\_state = rand\_state[pixel\_index];

vec3 col(0, 0, 0);

float u = float(i) / float(max\_x);

float v = float(j) / float(max\_y);

ray r = (\*cam)->get\_ray(u, v);

for (int t = 0; t < k; t++) //brute force k random directions to find the degree of darkening

col += color(r, world, &local\_rand\_state);

rand\_state[pixel\_index] = local\_rand\_state;

col /= float(k);

fb[pixel\_index] = col;

}

Данный метод автоматически распараллеливается при создании ядра. С помощью threadIdx,blockIdx.x,blockDim.x получаем индекс текущего пикселя для обработки. Далее мы запускаем луч от камеры и выполняем метод color. Итерация используется для большей точности финального результата.

В результате получаем цвет кажого пикселя для вывода. В качестве выходного формата изображения был использован ppm. Реализация вывода представлена ниже:

FILE \* f = fopen("cuda.ppm", "w");

fprintf(f, "P3\n%d %d\n%d\n", x, y, 255);

for (int j = y - 1; j >= 0; j--) {

for (int i = 0; i < x; i++) {

size\_t pixel\_index = j \* x + i;

int ir = int(255.99\*fb[pixel\_index].r());

int ig = int(255.99\*fb[pixel\_index].g());

int ib = int(255.99\*fb[pixel\_index].b());

fprintf(f, "%d %d %d ", ir, ig, ib);

}

}

В результате получаем следующее изображение:

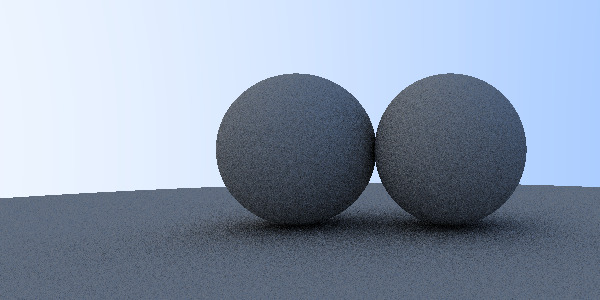


Рисунок 1.1 – пример работы программы.

Генерация данного изображения на GPU заняла 0.943 секунды, в то время как на CPU – 10.813 секунды. Для проверки работоспособности в данном случае использовался компьютер с видеокартой nvidia geforce 940mx. При использовании компьютера с видеокартой nvidia geforce gtx 1080ti генерация изображения заняла 0.435 секунды. При использовании более мощных видеокарт можно добиться еще более весомых результатов. По данным эксперементам видно, что распараллеливание имеет огромное преимущество при реализации технологии трассировки лучей. Именно поэтому многие графические задачи проводятся именно на видеокартах.

**Заключение**

В результате выполненной работы была рассмотрена архитектура CUDA, а также рассмотрены различные способы реализации метода трассировки лучей для графических задач.

Были проведены различные опыты для оценки продуктивности различных методов оптимизаций.

**Список использованной литературы**

# [1] CUDA: Как работает GPU - <https://habr.com/ru/post/54707/>

[2] Nvidia developer blog - <https://devblogs.nvidia.com/accelerated-ray-tracing-cuda/>