МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ

(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационные технологии и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

**Курсовой работа**

**по курсу «Параллельная обработка данных»**

**Обратная трассировка лучей (*Ray Tracing*).**

**Технологии *MPI*, *CUDA* и *OpenMP***

Выполнил: А. О. Дубинин

Группа: 8О-407Б

Преподаватели: К.Г. Крашенинников,

А.Ю. Морозов

Москва, 2021

**Условие**

**Цель работы**.

Использование *GPU* для создания фотореалистической визуализации. Рендеринг полузеркальных и полупрозрачных правильных геометрических тел. Получение эффекта бесконечности. Создание видеоролика.

Задание.

**Сцена.** Прямоугольная текстурированная поверхность (пол), над которой расположены три платоновых тела. Сверху находятся несколько источников света. На каждом ребре многогранника располагается определенное количество точечных источников света. Грани тел обладают зеркальным и прозрачным эффектом. За счет многократного переотражения лучей внутри тела, возникает эффект бесконечности.

**Камера.** Камера выполняет облет сцены согласно определенным законам. В

цилиндрических координатах () положение и точка направления камеры в

момент времени определяется следующим образом:

Где

.

Требуется реализовать алгоритм обратной трассировки лучей

( <http://www.ray-tracing.ru/> ) с использованием технологии *CUDA*. Выполнить покадровый рендеринг сцены. Для устранения эффекта «зубчатости», выполнить сглаживание (например, с помощью алгоритма *SSAA*). Полученный набор кадров склеить в видеоролик любым доступным программным обеспечением. Подобрать параметры сцены, камеры и освещения таким образом, чтобы получить наиболее красочный результат. Провести сравнение производительности *gpu* и *cpu* (т.е. дополнительно нужно реализовать алгоритм без использования *CUDA*).

Вариант 7.

На сцене должны располагаться три тела:

7. Гексаэдр, Октаэдр, Додекаэдр

Программное и аппаратное обеспечение

**GeForce 940MX**

|  |  |
| --- | --- |
| Compute capability: | 5.0 |
| Dedicated video memory: | 4096 MB |
| shared memory per block: | 49152 bytes |
| constant memory: | 65536 bytes |
| Total number of registers available per block: | 65536 |
| Maximum number of threads per multiprocessor: | 2048 |
| Maximum number of threads per block: | 1024 |
| ( 3) Multiprocessors, (128) CUDA Cores/MP: | 384 CUDA Cores |

**Intel(R) Core (TM) i5-7200U CPU @ 2.50GHz**

|  |  |
| --- | --- |
| Architecture: | x86\_64 |
| Byte Order: | Little Endian |
| CPU(s): | 4 |
| Thread(s) per core: | 2 |
| Core(s) per socket: | 2 |
| CPU MHz:  CPU max MHz:  CPU min MHz: | 713.848  3100,0000  400,0000 |
| L1d cache:  L1i cache:  L2 cache:  L3 cache: | 64 KiB  64 KiB  512 KiB  3 MiB |

|  |  |
| --- | --- |
| RAM | 8GiB SODIMM DDR4 Synchronous Unbuffered (Unregistered) 2400 MHz (0,4 ns) |

|  |  |
| --- | --- |
| SSD(SPCC\_M.2\_SSD) | 223,6G |
| HDD(ST1000LM035-1RK172) | 931,5G |

**OS: Ubuntu 20.04 focal**

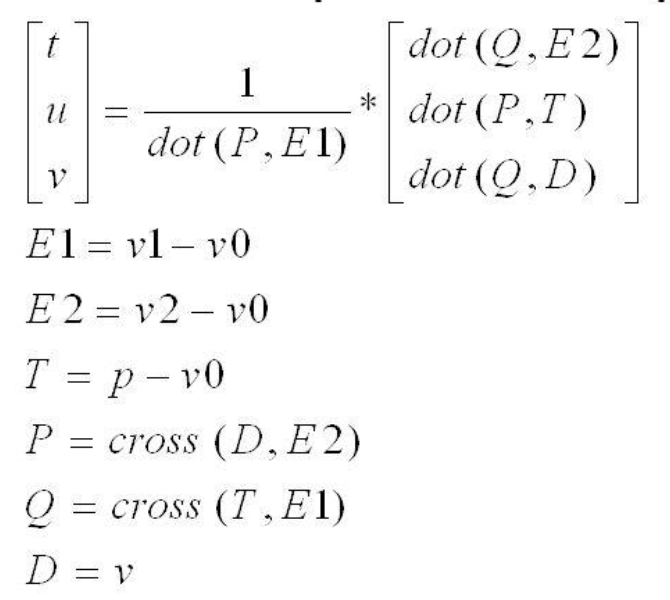
**IDE: jetbrains clion**

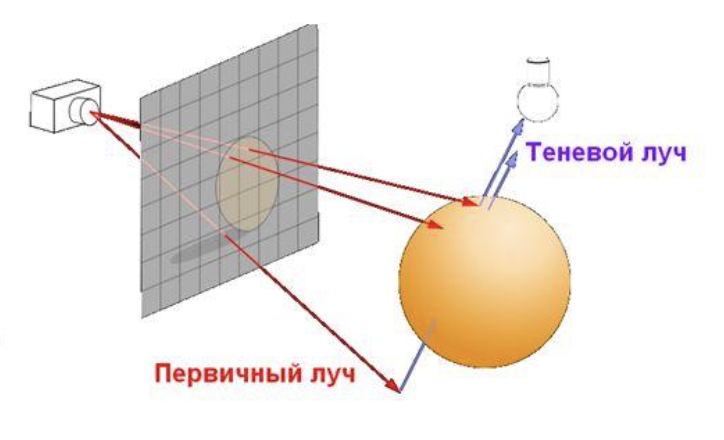
**compiler: nvcc**

Метод решения

**Алгоритм**

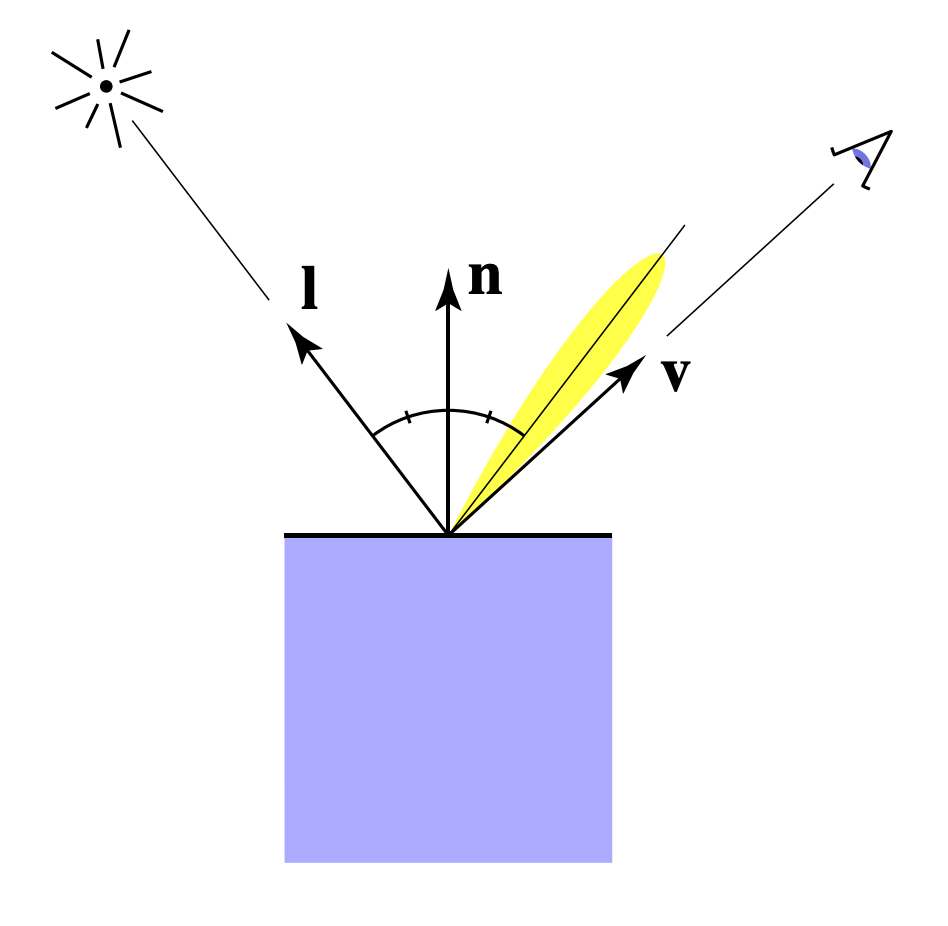
Поверхность задана, как массив треугольников. Главное вычисление происходит, при поиске пересечения луча и поверхности. Мы используем оптимизированный вариант вычислений, который был показан на лекции





**Свет**

Нужно было добавить тени в код лекции, для этого мы использовали модель освещения фонга. При вычислении нужно точки выпускаем луч в источник света, как мы делали в случае выпуска лучей из точки наблюдения. Если этот луч врежится в какую-либо поверхность до попадания в источник света, то эта точка будет являться тенью.

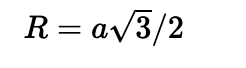


**Тела**

Самое сложное для отладки было нарисовать тела. Мой вариант предусматривал появление Додекаэдр. Начинал я с других фигур, где я выбрал кодстаил под названием hardcode, что казалось очень естественно, когда вершин фигур мало. Когда дело дошло до Додекаэдра я осознал, что мой кодстайл был не особо приемлимый, но все же я решил, продолжить отрисовку 30 полигонов. Ошибки были сложно отлавливаемый, но при рендеренге можно было понять примерно где искать.

Что касается самих тел:

Hexahedron – куб строися на основании формулы радиуса описанного шара, откуда вычислив сторону. Мы легко могли найти все углы путем сложения\вычитания половины стороны.



Octahedron – строился логически просто, координаты были взяты из википедии.

( ±1, 0, 0 );

( 0, ±1, 0 );

( 0, 0, ±1 ).

Dodecahedron – так же были взяты формулы из википедии для правильно додекаэдра. Формулы были взяты для радиуса , поэтому отнормировав для нашего радиуса и центра мы получили правильный додекаэдр

(±1, ±1, ±1)

(0, ±*ϕ*, ±1/ *ϕ*)

(±*1/ϕ*, 0, ±*ϕ*)

(±*ϕ*, ± *1/ϕ*, 0)

, где *ϕ – золотое сечение.*

**SSAA**

Метод позволяющий бороться с алисингом был взять из лр2, где текстурная ссылка успешно заменилась на обычную.

**for**(y = 0; y < h; y += 1) {  
 **for**(x = 0; x < w; x += 1) {  
 s = make\_uint4(0,0,0,0);  
 **for** (i = 0; i < wScale; ++i) {  
 **for** (j = 0; j < hScale; ++j){  
 p = src[ w \* wScale \* (y \* hScale + j)

+ (x \* wScale + i) ];  
 s.x += p.x;  
 s.y += p.y;  
 s.z += p.z;  
 }  
 }  
 s.x /= n;  
 s.y /= n;  
 s.z /= n;  
 out[y \* w + x] = make\_uchar4(s.x, s.y, s.z, s.w);  
 }  
}

**MPI**

Самое сложное было вспомнить как правильно пересылать динамические данные другим процессам, для этого был выбрал broadcast. Единственное изменение, кроме передачи параметров состояло в том, что каждый процесс вычисляет определенные кадры, для этого была изменена одна строка.

< for (int iter = id; iter < frames; iter += numproc) {  
---  
> for (int iter = 0; iter < frames; ++iter) {

**OpenMP**

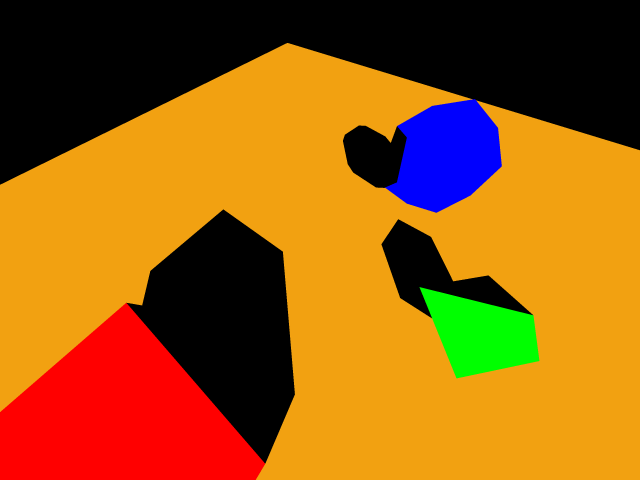
Было добавлено распараллеливание рендера на cpu:

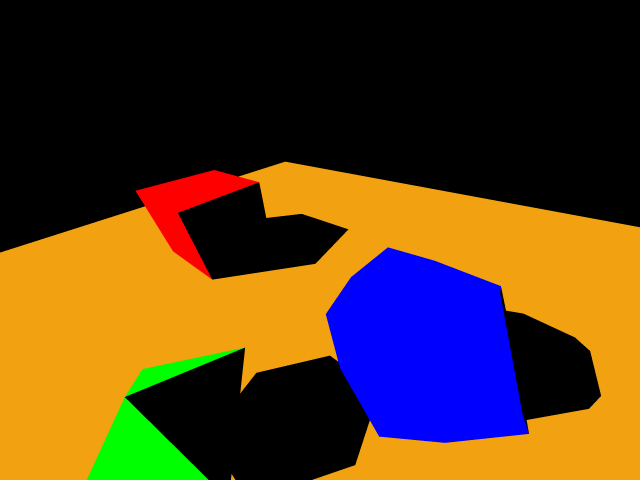
#pragma omp parallel **for**

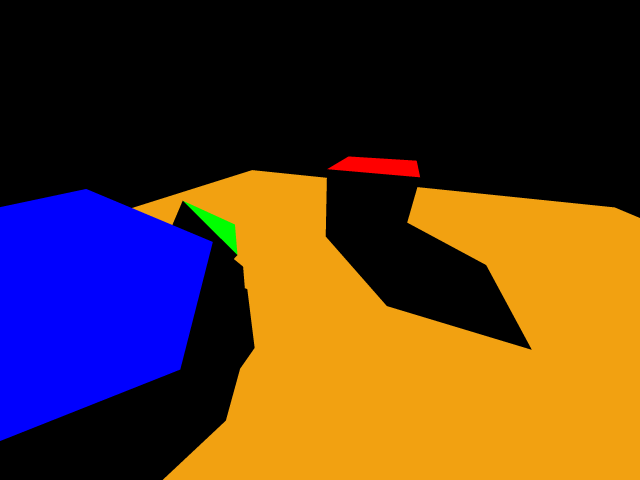
Так же был вариант добавить прагму в SSAA, но после некоторых проверок на скорость работы, это решение было отметено.

Результат

Конфигурация сцены была выбрана методом подбора поэтому пролет не получился идеальным. Создание видео было выполнено с помощью gimp, как было показано на лекции







Сравним скорость обработки ray tracing, на cpu и gpu. Сравнивать мы будем, меняя коэффициент SSAA, так как при большом коэффициенте будет генерироваться большая картинка и соответственно будет выпускаться больше лучей. Картинка на выходе будет иметь размер 640x480. Будем считать среднее время, затрачиваемое на генерацию одного кадра.

|  |  |  |  |
| --- | --- | --- | --- |
| SSAA multiplier | Rays | GPU (ms) | CPU (ms) |
| 1 | 307200 | 61.1 | 1284.3 |
| 2 | 1228800 | 225.6 | 5618.4 |
| 4 | 4915200 | 819.4 | 21557.1 |

Сравним скорость работы на CPU с openmp и без него

|  |  |  |  |
| --- | --- | --- | --- |
| SSAA multiplier | Rays | Openmp | CPU (ms) |
| 1 | 307200 | 313.5 | 1284.3 |
| 2 | 1228800 | 1213.5 | 5618.4 |
| 4 | 4915200 | 4943.5 | 21557.1 |

Скорость работы c mpi и без него сложно мерить, так как в моем распоряжении нет настоящего кластера и так как затраты на поддержку процессов и broadcast большие. Так же есть вероятность, что на сервере, который мне для моего пользователя стоит ограничения на cpu, поэтому код mpi процессов выполняется на одном физическом потоке. Поэтому время с одним процессом и с множеством не сильно отличается.

В данном случае в графе время указывалось общее время работы программы.

|  |  |  |  |
| --- | --- | --- | --- |
| SSAA multiplier | Rays | Np = 8 | Np = 1 |
| 4 | 4915200 | 0m27.589s | 0m27.175s |
| 8 | 19660800 | 1m58.653s | 1m46.824s |

Выводы

Исходя из результатов мы можем понять, что openmp сильно ускоряет работу с cpu, а интерфейс работы с openmp попрежнему сохраняет лидерство в моем сердце по своей простоте. Работа с MPI была интересна с точки зрения запуска и анализа эффективности, хоть мне и не удалось получить прироста производительности, было интересно наблюдать в htop, как все ядра системы были заняты моей программой.