

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*) на *GPU***

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 полужеркальных 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  $(r, \varphi, z)$  положение и точка направления камеры в

момент времени  $t$  определяется следующим образом:

$$r_c(t) = r_c^0 + A_c^r \sin(\omega_c^r \cdot t + p_c^r);$$

$$z_c(t) = z_c^0 + A_c^z \sin(\omega_c^z \cdot t + p_c^z);$$

$$\varphi_c(t) = \varphi_c^0 + \omega_c^\varphi t;$$

$$r_n(t) = r_n^0 + A_n^r \sin(\omega_n^r \cdot t + p_n^r);$$

$$z_n(t) = z_n^0 + A_n^z \sin(\omega_n^z \cdot t + p_n^z);$$

$$\varphi_n(t) = \varphi_n^0 + \omega_n^\varphi t,$$

Где

$$t \in [0, 2\pi].$$

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 *gpi* и *cpi* (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:	713.848
CPU max MHz:	3100,0000
CPU min MHz:	400,0000
L1d cache:	64 KiB
L1i cache:	64 KiB
L2 cache:	512 KiB
L3 cache:	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

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\text{dot}(P, E1)} * \begin{bmatrix} \text{dot}(Q, E2) \\ \text{dot}(P, T) \\ \text{dot}(Q, D) \end{bmatrix}$$

$$E1 = v1 - v0$$

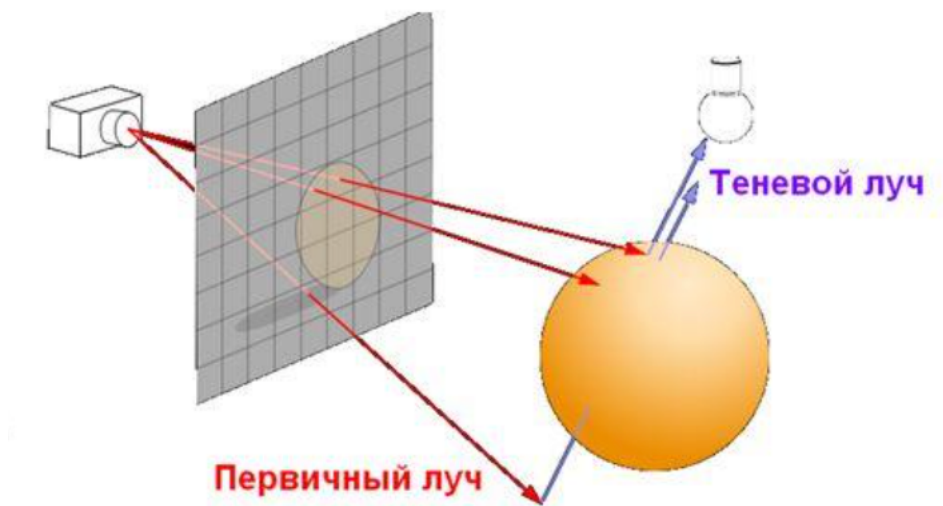
$$E2 = v2 - v0$$

$$T = p - v0$$

$$P = \text{cross}(D, E2)$$

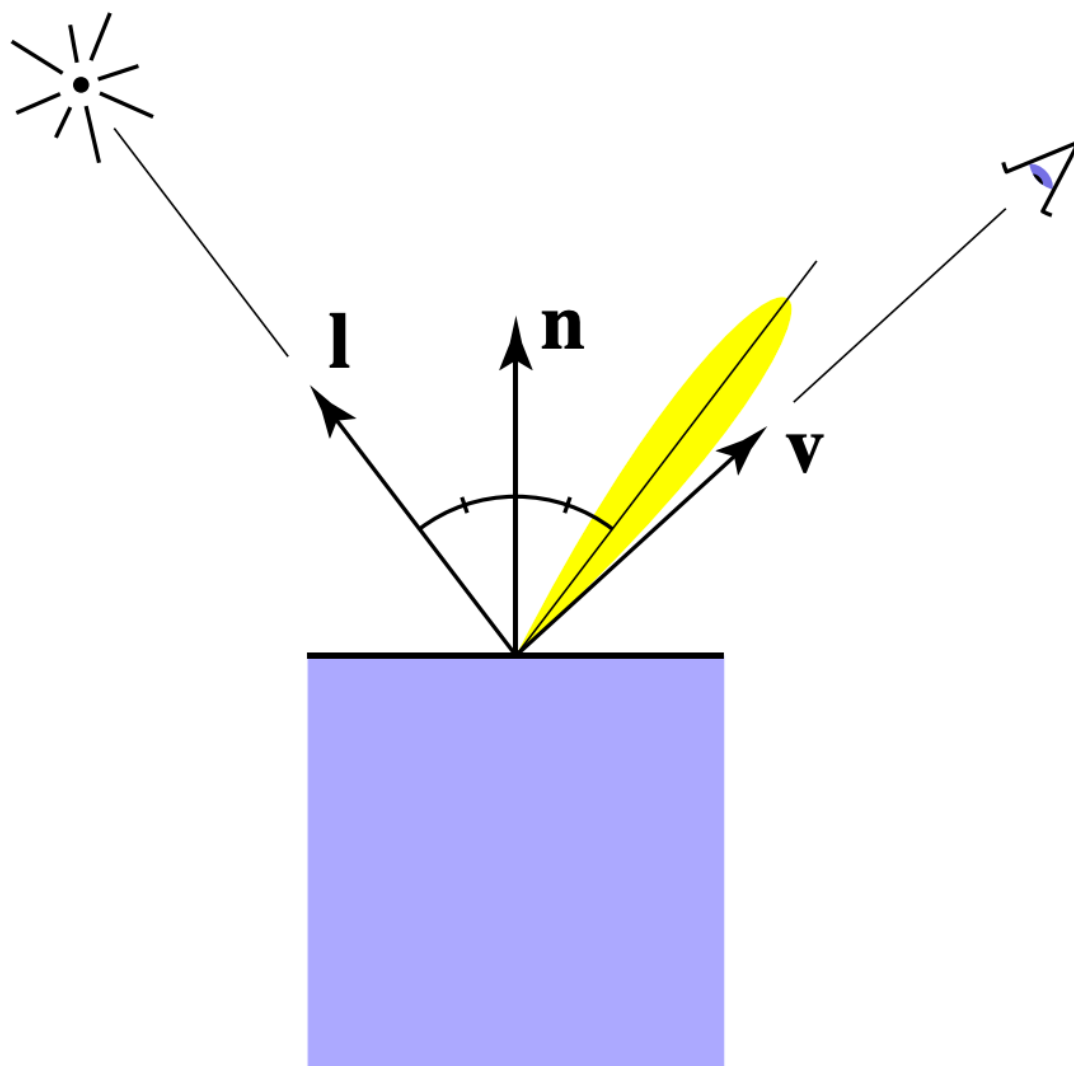
$$Q = \text{cross}(T, E1)$$

$$D = v$$



## Свет

Нужно было добавить тени в код лекции, для этого мы использовали модель освещения фона. При вычислении нужно точки выпускаем луч в источник света, как мы делали в случае выпуска лучей из точки наблюдения. Если этот луч врежется 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 кодстайл 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 строился на основании формулы радиуса описанного шара, откуда вычислив сторону. Мы легко могли найти все углы путем сложения\вычитания половины стороны.

$$R = a\sqrt{3}/2$$

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

(  $\pm 1$ , 0, 0 );

( 0,  $\pm 1$ , 0 );

( 0, 0,  $\pm 1$  ).

Dodecahedron – так же были взяты формулы из википедии для правильно додекаэдра. Формулы были взяты для радиуса  $\sqrt{3}$ , поэтому отнормировав для нашего радиуса и центра мы получили правильный додекаэдр

( $\pm 1$ ,  $\pm 1$ ,  $\pm 1$ )

(0,  $\pm \phi$ ,  $\pm 1/\phi$ )

( $\pm 1/\phi$ , 0,  $\pm \phi$ )

( $\pm \phi$ ,  $\pm 1/\phi$ , 0)

, где  $\phi$  – золотое сечение.

## SSAA

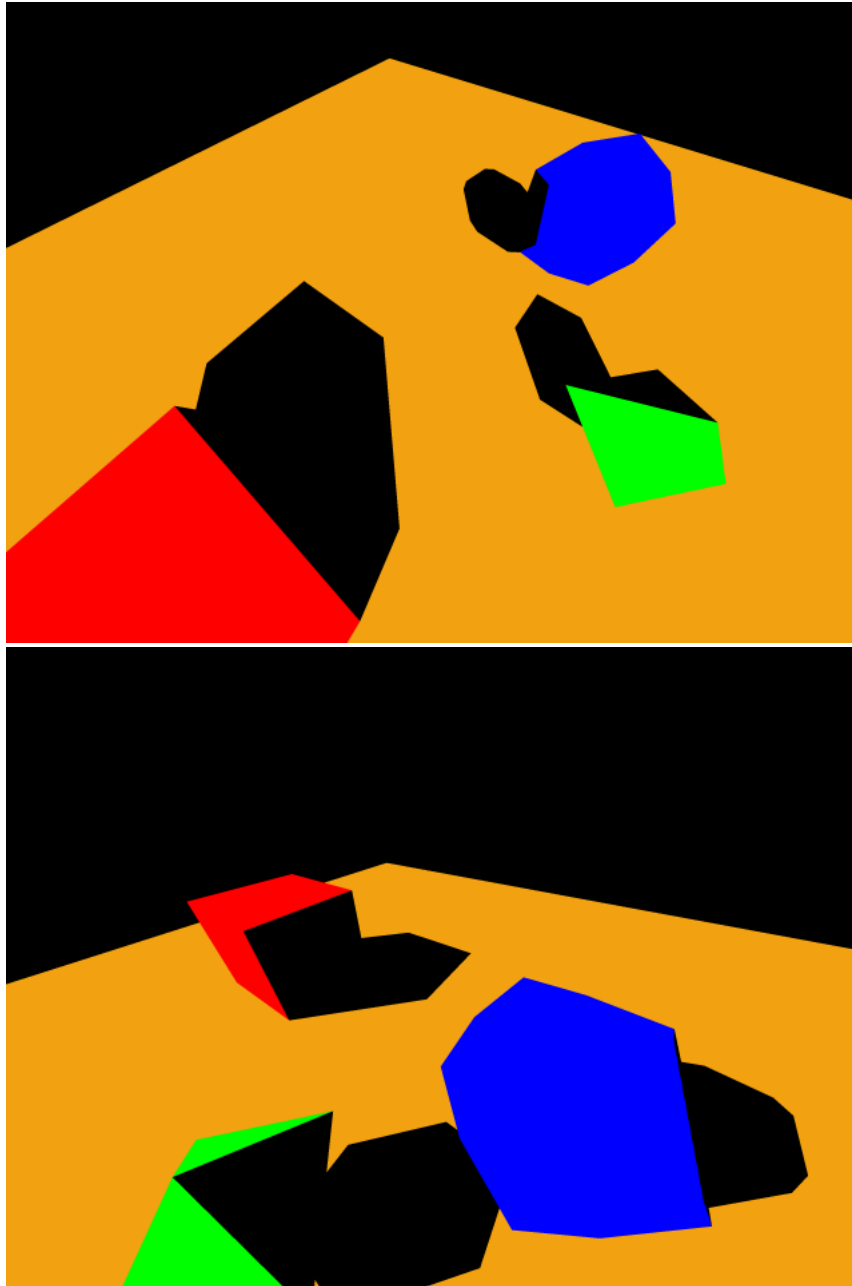
Метод позволяющий бороться с алисингом был взят из лр2, где текстурная ссылка успешно заменилась на обычную.

```
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) ];

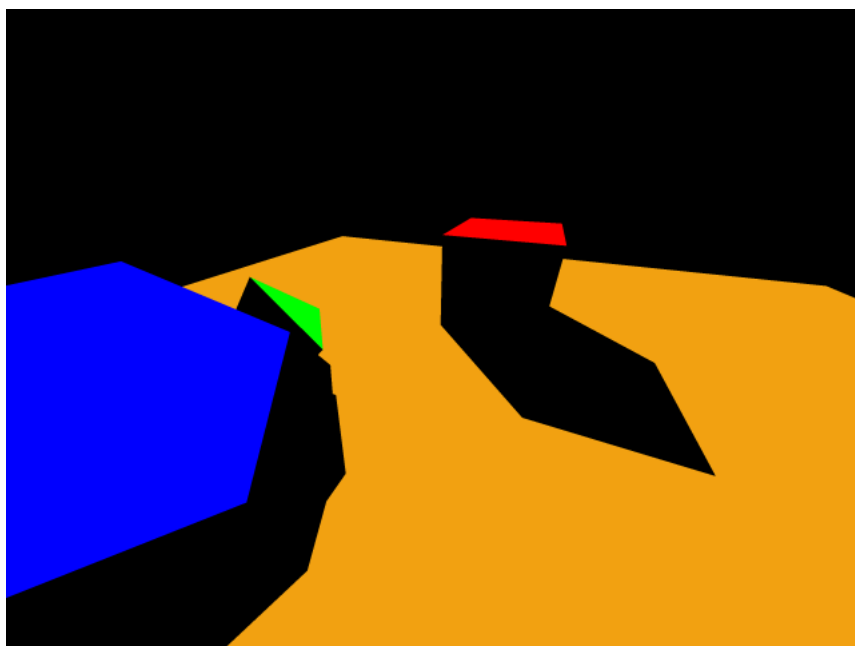
                s.x += p.x;
                s.y += p.y;
                s.z += p.z;
            }
        }
        s.x /= n;
        s.y /= n;
        s.z /= n;
        out[y * w + x] = make_uchar4(s.x, s.y, s.z, s.w);
    }
}
```

## 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. Сравнить 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

## **Выводы**

Исходя из результатов мы можем понять, что неприемлемо долго выполняется ray tracing на cpu, учитывая не самую сложную модель данных. Поэтому говорить о raytracing в runtime на cpu можно только мечтать. Как мы знаем с недавних пор у новых видеокарт присутствуют аппаратная поддержка raytracing, поэтому знание cuda очень поможет тому, кому хочется развиваться в данной области. Данная лабораторная работа мне понравилась, так как с данной лр я узнал гораздо больше, чем со всего курса компьютерной графики.