МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»
Кафедра №806 «Вычислительная математика и программирование»

Курсовая работа
по курсу «Параллельная обработка данных»

Обратная трассировка лучей (Ray Tracing) на GPU

Выполнил: Е.С. Пищик
Группа: 8О-406Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2022

Условие

Цель работы. Использование GPU для создание фотореалистической визуализации. Рендеринг полузеркальных и полупрозрачных правильных геометрических тел. Получение эффекта бесконечности. Создание анимации.

Вариант 8: Гексаэдр, Октаэдр, Икосаэдр.

Программное и аппаратное обеспечение

GPU via SSH:

- Compute capability: 2.1
- Name: GeForce GT 545
- Total Global Memory: 3150381056
- Shared memory per block: 49152
- Registers per block: 32768
- Warp size: 32
- Max threads per block: (1024, 1024, 64)
- Max block: (65535, 65535, 65535)
- Total constant memory: 65536
- Multiprocessors count: 3

CPU via SSH: Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
RAM via SSH: 16 Gb.
HDD via SSH: 471 Gb.
OS: Windows 10
OS via SSH: Ubuntu 16.04.6 LTS
IDE: Visual Studio Code
Compiler via SSH: nvcc

Метод решения

Ray Tracing:

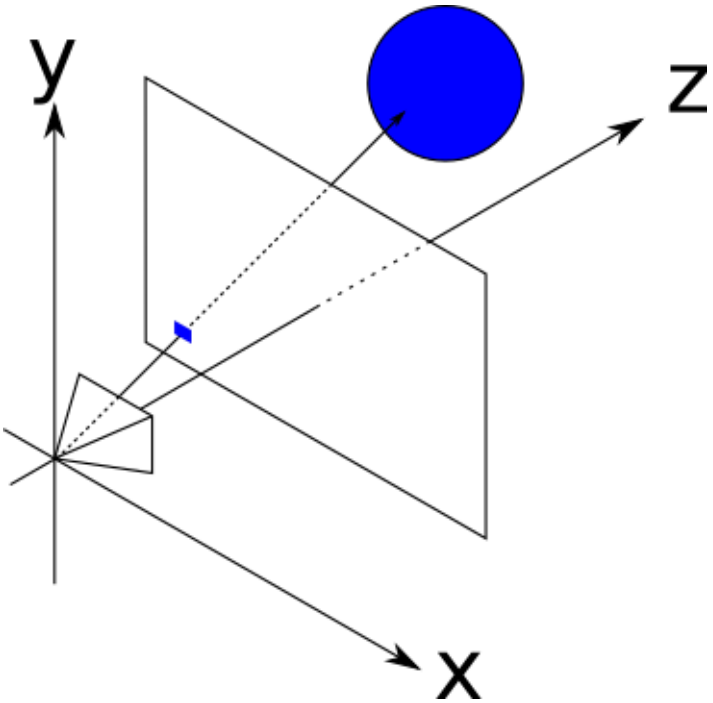


Рис. 1. Трёхмерная графика с нуля. Часть 1: трассировка лучей.

Из каждой точки экрана мы выпускаем луч, т.к. мы дополнительно используем алгоритм SSAA, то количество лучей увеличивается в квадрат коэффициента SSAA. Был выполнен вариант на “три” с нулевым уровнем рекурсии, без отражений, без текстур, простые модели без отдельных ребер с источниками света, один источник света. Т.к. уровень рекурсии нулевой, отражения лучей не происходят, мы запускаем лучи из экрана на сцену и проверяем пересечение с фигурами, если луч пересекается с фигурой, то происходит отображение. Вычисление тени происходит следующим образом - выпускаем луч из точки в камеру, если луч пересекает фигуру, то это теневой луч.

SSAA:

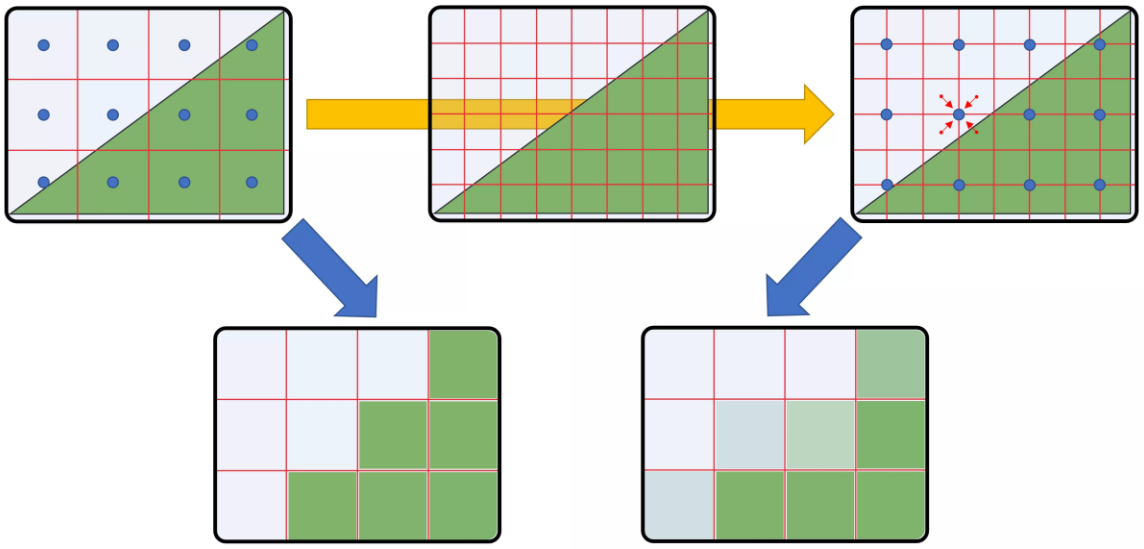


Рис. 2. Как работает рендеринг 3D-игр: сглаживание с помощью SSAA, MSAA, FXAA, TAA и других методик.

При помощи коэффициента SSAA вычисляем размер новой области, который в квадрат коэффициента SSAA больше, чем размер исходной области, дальше происходит рендеринг новой области, проход по ней и усреднение значений соседних пикселей, таким образом получаем картинку исходного разрешения, но с более плавными границами объектов.

Фигуры:

Гексаэдр - по-другому куб, фигура состоящая из 6 граней, каждая из которых правильный четырехугольник. Каждая грань состоит из двух полигонов (треугольников), составляем полигоны из координат вершин куба - сначала для куба вписанного в сферу радиуса 1 с центром в точке 0, а потом домножаем значения наших координат полигона на значения радиуас сферы для соответствующей оси и смещаем значения координат полигона на значения координат центра фигуры.

Октаэдр - фигура, состоящая из 8 правильных треугольников, задаем координаты 6 вершин расстояние от центра фигуры до каждой из вершин равно радиусу описанной сферы, отсюда можно задать координаты всех 6 вершин, если центр октаэдра расположен в точке (0, 0, 0) - каждая вершина имеет два нулевых значения на двух из трех осей, а значения одной из осей равно радиусу сферы описанной вокруг октаэдра. После вычисления данных координат смещаем каждое значение на соответствующее значение центра октаэдра.

Икосаэдр - фигура, состоящая из 20 правильных треугольников. Чтобы построить данную можно использовать сферические координаты - цитата из wikipedia “ если

считать, что две вершины находятся на северном и южном полюсах (широта ±90 °), то остальные десять вершин находятся на широте ± арктангенс(0.5) = ±26.57°. Эти десять вершин расположены на равном расстоянии друг от друга (36 ° друг от друга), чередуясь между северными и южными широтами”.

Описание программы

Программа состоит из одного файла - render.cu, функции icosa, hexa, octa используются для построения соответствующих фигур, в функции figures мы строим эти три фигуры. В функции create_scene мы создаем саму сцену, в частности пол, который состоит из двух полигонов. Функции с приставкой ssaa - реализуют алгоритм SSAA на cpu и gpu. Функции с постфиксом _render - занимаются рендерингом сцены на cpu и gpu. Функция ray - отвечает за обработку одного луча. В функции main собираем все части нашей программы, считываем данные, записываем результат.
Основные ядра - ssaa_gpu, gpu_render, ray.

SSAA:

```
__global__ void ssaa_gpu(uchar4 *in_data,
                         uchar4 *out_data,
                         int w,
                         int h,
                         int k)
{
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   int idy = blockIdx.y * blockDim.y + threadIdx.y;

   int offsetx = blockDim.x * gridDim.x;
   int offsety = blockDim.y * gridDim.y;

   for (int y = idy; y < h; y += offsety)
   {
      for (int x = idx; x < w; x += offsetx)
      {
         int4 mid = {0, 0, 0, 0};
         for (int j = 0; j < k; ++j)
         {
            for (int i = 0; i < k; ++i)
            {
               int ind = k * k * y * w + k * j * w + k * x + i;
               mid.x += in_data[ind].x;
               mid.y += in_data[ind].y;
               mid.z += in_data[ind].z;
            }
         }
```

```
            double norm = k * k;
            out_data[x + y * w] = make_uchar4(mid.x / norm,
                                              mid.y / norm,
                                              mid.z / norm,
                                              0);
        }
    }
}
```

Render:

```
__global__ void gpu_render(vec3 pc,
                           vec3 pv,
                           int w,
                           int h,
                           double angle,
                           uchar4 *data,
                           vec3 light_pos,
                           vec3 light_clr,
                           triangle *trgs,
                           int ssaa_sqrt)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    int idy = blockDim.y * blockIdx.y + threadIdx.y;
    int offsetx = blockDim.x * gridDim.x;
    int offsety = blockDim.y * gridDim.y;

    double dw = 2.0 / (w - 1.0);
    double dh = 2.0 / (h - 1.0);
    double dz = 1.0 / tan(angle * M_PI / 360.0);

    vec3 bz = normalize(pv - pc);
    vec3 bx = normalize(prod(bz, {0.0, 0.0, 1.0}));
    vec3 by = normalize(prod(bx, bz));

    for (int x = idx; x < w; x += offsetx)
        for (int y = idy; y < h; y += offsety)
        {
            vec3 v = {-1.0 + dw * x, (-1.0 + dh * y) * h / w, dz};
            vec3 dir = mult(bx, by, bz, v);
            data[(h - 1 - y) * w + x] = ray(pc,
                                            normalize(dir),
                                            light_pos,
                                            light_clr,
                                            trgs,
```

```
                                    ssaa_sqrt);
        }
}
```

Ray Tracing:

```
#define raycast()                   \
    vec3 e1 = trgs[k].v2 - trgs[k].v1; \
    vec3 e2 = trgs[k].v3 - trgs[k].v1; \
    vec3 p = prod(dir, e2);          \
    double div = p * e1;             \
    if (fabs(div) < 1e-10)           \
        continue;                    \
    vec3 t = pos - trgs[k].v1;       \
    double u = (p * t) / div;        \
    if (u < 0.0 || u > 1.0)          \
        continue;                    \
    vec3 q = prod(t, e1);            \
    double v = (q * dir) / div;      \
    if (v < 0.0 || v + u > 1.0)      \
        continue;

__host__ __device__ uchar4 ray(vec3 pos,
                               vec3 dir,
                               vec3 light_pos,
                               vec3 light_clr,
                               triangle *trgs,
                               int ssaa_sqrt)
{
    int k_min = -1;
    double ts_min;

    for (int k = 0; k < ssaa_sqrt; ++k)
    {
        raycast();

        double ts = (q * e2) / div;
        if (ts < 0.0)
            continue;

        if (k_min == -1 || ts < ts_min)
        {
            k_min = k;
            ts_min = ts;
        }
```

```
    }

    if (k_min == -1)
        return {0, 0, 0, 0};

    pos = dir * ts_min + pos;
    dir = light_pos - pos;

    double size = sqrt(dir * dir);

    dir = normalize(dir);
    for (int k = 0; k < ssaa_sqrt; ++k)
    {
        raycast();

        double ts = (q * e2) / div;
        if (ts > 0.0 && ts < size && k != k_min)
            return {0, 0, 0, 0};
    }

    uchar4 color_min;
    color_min.x = trgs[k_min].clr.x;
    color_min.y = trgs[k_min].clr.y;
    color_min.z = trgs[k_min].clr.z;

    color_min.x *= light_clr.x;
    color_min.y *= light_clr.y;
    color_min.z *= light_clr.z;
    color_min.w = 0;

    return color_min;
}
```

Исследовательская часть и результаты

Все тесты проводились с данными входными параметрами, менялись только параметры ядра, коэффициент SSAA, размер изображения:

```
1
./out /img_%d.data
640 480 120
7.0 3.0 0.0 2.0 1.0 2.0 6.0 1.0 0.0 0.0
2.0 0.0 0.0 0.5 0.1 1.0 4.0 1.0 0.0 0.0
4.0 0.0 0.0 1.0 0.7 0.0 1.5 0.0 0.0 0.0
0.75 1.75 0.0 0.7 0.0 1.0 1.0 0.0 0.0 0.0
```

-4.0 -1.5 0.0 0.0 0.5 0.0 0.8 0.0 0.0 0.0
-10.0 -10.0 -1.0 -10.0 10.0 -1.0 10.0 10.0 -1.0 10.0 -10.0 -1.0 none 0.3 0.3 0.3 0.25
1
25 25 25 1.0 1.0 1.0
1 5

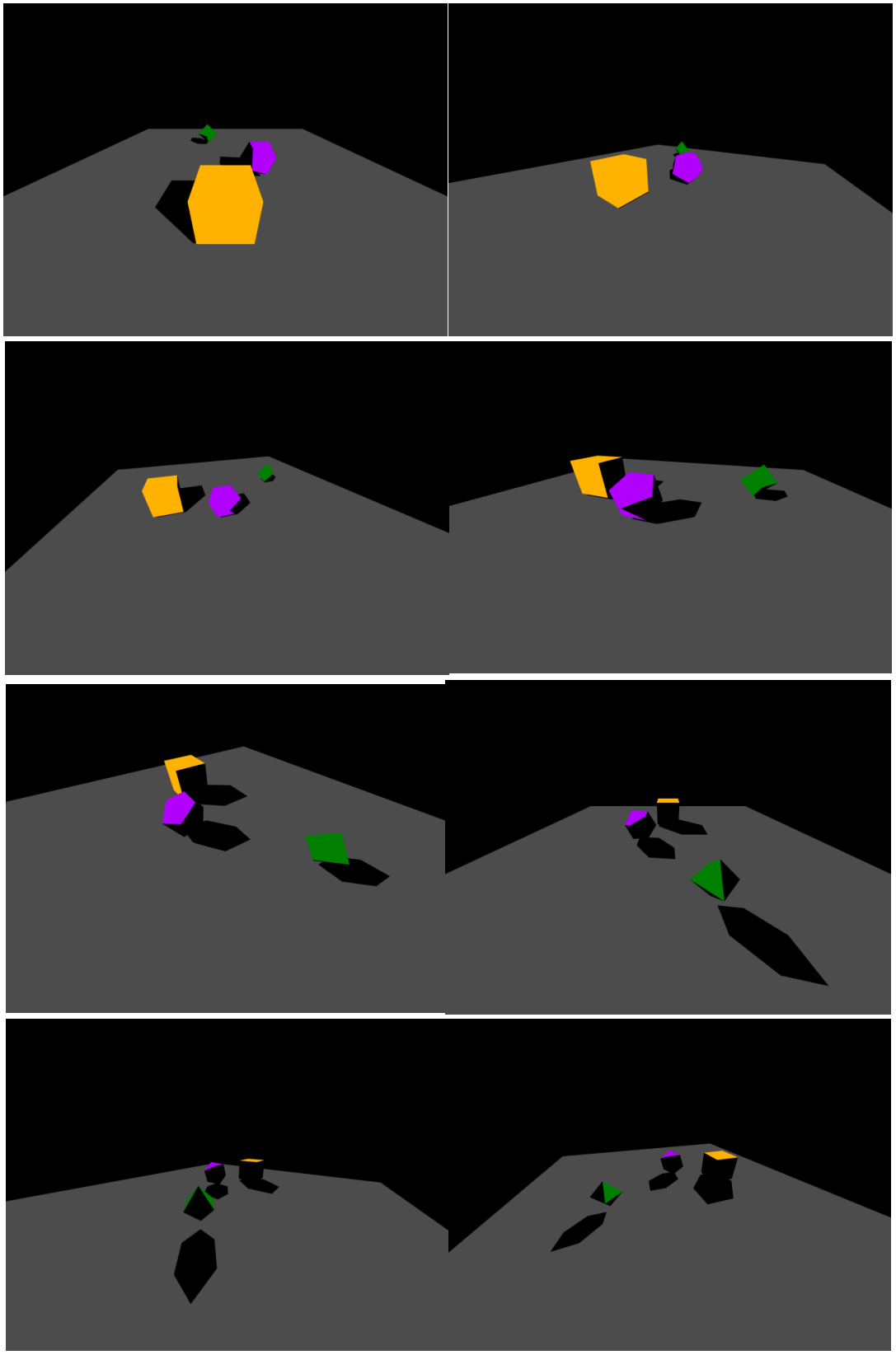
Время на обработку одного кадра на GPU с различными параметрами ядер и изображений, ms.

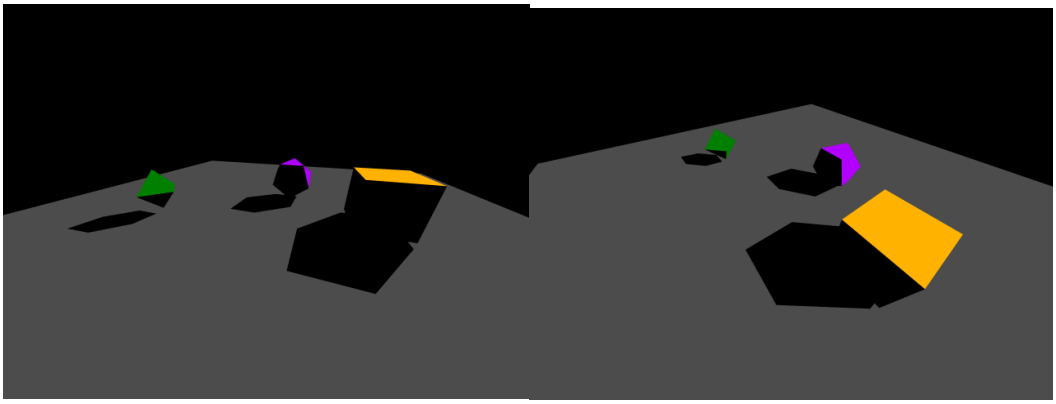
	<<<64,64>>>	<<<128,128>>>	<<<256,256>>>	<<<512,512>>>
640 * 480 * 3 * 3	620	622	679	747
640 * 480 * 5 * 5	1352	1395	1406	1451
640 * 480 * 9 * 9	3824	3798	3974	3967
1366 * 768 * 3 * 3	1637	1621	1716	1734
1366 * 768 * 5 * 5	4106	4066	4036	4393
1366 * 768 * 9 * 9	13207	12692	12724	12800
1920 * 1080 * 3 * 3	2966	2941	3064	3093
1920 * 1080 * 5 * 5	8042	7719	7920	8192
1920 * 1080 * 7 * 7	15672	14888	15165	15124

Время на обработку одного кадра на GPU против CPU с различными параметрами изображений, ms.

	GPU <<<128, 128>>>	CPU
320 * 240 * 5 * 5	517	5805
320 * 240 * 7 * 7	771	11236
640 * 480 * 5 * 5	1396	23070
640 * 480 * 7 * 7	2439	44717

Красивые изображения были получены при помощи следующей конфигурации на GPU<<<128,128>>> из 100 изображений были выбраны 10 с различных ракурсов.
100
./out /img_%d.data
640 480 120
7.0 3.0 0.0 2.0 1.0 2.0 6.0 1.0 0.0 0.0
2.0 0.0 0.0 0.5 0.1 1.0 4.0 1.0 0.0 0.0
4.0 0.0 0.0 1.0 0.7 0.0 1.5 0.0 0.0 0.0
0.75 1.75 0.0 0.7 0.0 1.0 1.0 0.0 0.0 0.0
-4.0 -1.5 0.0 0.0 0.5 0.0 0.8 0.0 0.0 0.0
-10.0 -10.0 -1.0 -10.0 10.0 -1.0 10.0 10.0 -1.0 10.0 -10.0 -1.0 none 0.3 0.3 0.3 0.25
1
25 25 25 1.0 1.0 1.0
1 5





Выводы

В данной курсовой работе я изучил алгоритм обратной трассировки лучей (Ray Tracing) на GPU, данная технология может применяться для создания красивых, фотореалистичных изображений, красивой графики в играх. Главный недостаток такого подхода - высокая ресурсозатратность, нужны очень мощные GPU для адекватной скорости обработки кадра. Данная технология все более активно используется в различных задачах (например в играх), т.к. в настоящее время активно появляются все более мощные видеокарты. Самой сложной частью было программирование функции выпускания луча, т.к. ранее я не очень хорошо представлял, как это программируется, также данная функция использует некоторые математические идеи, которые я до этого не знал, но почитав источники и посмотрев лекцию я смог разобраться в данном алгоритме. Проведя различные тесты мы видим преимущество использования GPU в данной задаче, причем чем больше разрешение изображения и коэффициент SSAA, т.е. чем больше лучей нам нужно обработать, тем сильнее CPU проигрывает GPU.

Литература

1. Трёхмерная графика с нуля. Часть 1: трассировка лучей.
2. Как работает рендеринг 3D-игр: сглаживание с помощью SSAA, MSAA, FXAA, TAA и других методик.
3. Гексаэдр, Октаэдр, Икосаэдр.