МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

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

Курсовой работа
по курсу «Параллельная обработка данных»

Обратная трассировка лучей (Ray Tracing) на GPU

Выполнил: К.С. Чистяков
Группа: 8О-406Б-21
Преподаватель: А.Ю. Морозов

Москва, 2025

Условие

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

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, φ, 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 с использованием технологии CUDA. Выполнить покадровый рендеринг сцены. Для устранения эффекта «зубчатости», выполнить сглаживание (например с помощью алгоритма SSAA). Полученный набор кадров склеить в анимацию 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).

2. **Вариант задания.** Вариант 1. На сцене должны располагаться три тела: тетраэдр, гексаэдр, октаэдр.

Программное и аппаратное обеспечение

Графический процессор (Google Colab)

Compute capability	7.5

Name	Tesla T4
Total Global Memory	15828320256
Shared memory per block	49152
Registers per block	65536
Warp size	32
Max threads per block	(1024, 1024, 64)
Max block	(2147483647, 65535, 65535)
Total constant memory	6553
Multiprocessors count	40

Процессор AMD Ryzen 5 4500U with Radeon Graphics (2.38 GHz)

Technology	7 nm
Cores	6
Threads	6
Core Speed	1390 MHz
Cache (L1 Data)	6 x 32 KBytes (8-way)
Cache (L1 Inst.)	6 x 32 KBytes (8-way)
Cache (Level 2)	6 x 512 KBytes (8-way)
Cache (Level 3)	2 x 4 MBytes (16-way)

Оперативная память

Number of modules	2
Module Manuf.	Samsung
Module Size	8 GBytes (total - 16 GBytes)
Generation	DDR4

Жесткий диск - 512 ГБ SSD
OS - Windows 10 Домашняя + Linux

IDE - VS Code + Google Colab
Compiler - nvcc, gcc

Метод решения

Инициализация сцены:

- задание параметров сцены, включая источники света, геометрию объектов, параметры камеры и настройки трассировки лучей;
- создание фигур (гектаэдр, тетраэдр, октаэдр) и их масштабирование/смещение согласно заданным параметрам.

Выбор метода рендеринга:

- чтение входных данных и флагов командной строки, определяющих режим работы;
- инициализация необходимых ресурсов на CPU/GPU.

Генерация изображения:

Для каждого пикселя изображения:

- вычисляется направление луча;
- запускается трассировка луча с учетом пересечений с объектами сцены;
- определяется цвет пикселя с учетом модели освещения.

Трассировка лучей:

Определение пересечения луча с объектами сцены.

Если пересечение найдено:

- вычисление нормали в точке пересечения;
- вычисление освещенности с учетом затенения и отражений;
- расчет цвета пикселя.

Формирование итогового изображения:

- вычисленные пиксели записываются в буфер кадра;
- буфер сохраняется в файл в бинарном формате.

Описание программы

Программа содержит один файл.

- **#define CSC(stmt)** - функция для отлова ошибок CUDA

- **struct Vector3** - структура представляющая 3Д вектор, включает методы для арифметических операций, вычисления длины, нормализации, скалярного и векторного произведений
- **struct ColorRGBA** - структура представляющая цвет в формате RGBA
- **struct TriPrim** - структура представляющая треугольник, включает метод для вычисления нормали треугольника
- **struct Illumination** - структура представляющая источник света
- **struct HitRecord** - структура для хранения информации о пересечении луча с объектом сцены
- **struct SegmentGlow** - структура определяющая параметры свечения по сегментам, включая начало и конец свечения
- **struct SceneObject** - структура представляющая объект сцены
- **__host__ __device__ bool checkAxisAlignment(const Vector3& vA, const Vector3& vB)** - функция проверяющая, выровнены ли две точки по одной оси
- **static std::array<TriPrim, 12> createCube()** - задаём параметры куба (гексаэдра)
- **static std::array<TriPrim, 4> createTetra()** - задаём параметры тетраэдра
- **static std::array<TriPrim, 8> createOcta()** - задаём параметры октаэдра
- **template<size_t N> void shiftAndScalePrimitives(std::array<TriPrim, N>& arr, const Vector3& offset, double scaleVal)** - функция для смещения и масштабирования объекта в пространстве
- **__host__ __device__ HitRecord checkHits(const Vector3& orig, const Vector3& dir, double minT, double maxT, const SceneObject* shapes, size_t shapeCount)** - функция определяющая, пересекает ли луч (заданный начальной точкой и направлением) какой-либо объект в сцене
- **__host__ __device__ ColorRGBA getShadingColor(const SceneObject& obj, const Illumination& lamp, const Vector3& hitPnt, const Vector3& viewRay, const SceneObject* shapes, size_t shapeCount)** - функция вычисляющая цвет точки пересечения луча с объектом, включая освещение, затенение и эффекты прозрачности
- **__host__ __device__ ColorRGBA rayTrace(const Vector3& origin, const Vector3& dir_, double tLow, double tHigh, const SceneObject* shapes, size_t shapeCount, const Illumination& lamp, int depth)** - функция рализующая трассировку лучей с учётом отражений, прозрачности и затенения
- **__host__ __device__ ColorRGBA getReflectionColor(const Vector3& o_, const Vector3& d_, double tLow, double tHigh, const SceneObject* shapes, size_t shapeCount, const Illumination& lamp, int depth)** - функция, рассчитывающая цвет отражённого луча
- **struct FramePixel** - структура использующаяся для представления пикселя кадра
- **__global__ void deviceRenderKernel(FramePixel* outFB, int w_, int h_, const SceneObject* shapes, int shapeCount, Illumination lamp, double fovVal, Vector3 camLoc, Vector3 lookAt)** - функция рендеринга на GPU, вычисляющая цвет каждого пикселя

- **void hostRender(FramePixel* fbPtr, int w_, int h_, const SceneObject* shapes, int shapeCount, Illumination lamp, double fovVal, Vector3 camLoc, Vector3 lookAt)** - функция рендеринга на CPU
- **struct ProgramInput** - структура для всех вводимых данных (флаг, параметры камеры, параметры света, три позиции и размеры объектов)
- **bool parseCmdFlags(int argc, char** argv)** - функция разбирающая аргументы командной строки (--default, --cpu, --gpu)
- **ProgramInput retrieveSceneInput(bool runOnGPU)** - функция считывающая входные данные сцены
- **std::vector<SceneObject> assembleScene(const ProgramInput& inputData)** - функция создающая сцену, состоящую из объектов
- **void hostRenderSequence(const ProgramInput& pp, const std::vector<SceneObject>& worldShapes)** - функция выполняющая всё, что нужно для рендеринга на CPU
- **void deviceRenderSequence(const ProgramInput& pp, const std::vector<SceneObject>& worldShapes)** - функция выполняющая всё, что нужно для рендеринга на GPU
- **int main(int argc, char** argv)** - главная функция (вызывает функции для разбора аргументов командной строки, загрузки параметров сцены, создания списка объектов сцены и непосредственно рендеринга на CPU или GPU

Ядро для рендеринга на GPU:

```
__global__ void deviceRenderKernel(FramePixel* outFB,
  int w_, int h_,
  const SceneObject* shapes,
  int shapeCount,
  Illumination lamp,
  double fovVal,
  Vector3 camLoc,
  Vector3 lookAt)
{
  double aspect = double(h_) / double(w_);
  double dx_ = 2.0 / (w_ - 1.0);
  double dy_ = 2.0 / (h_ - 1.0);
  double distPlane = 1.0 / tan(fovVal / 2.0);

  Vector3 fwd = lookAt.minus(camLoc).normalized();
  Vector3 upv(0, 0, 1);
  Vector3 rght = fwd.crossProd(upv).normalized();
  Vector3 trueUp = rght.crossProd(fwd).normalized();

  double bigVal = 1e20;
  int ix_ = blockIdx.x * blockDim.x + threadIdx.x;
  int iy_ = blockIdx.y * blockDim.y + threadIdx.y;
```

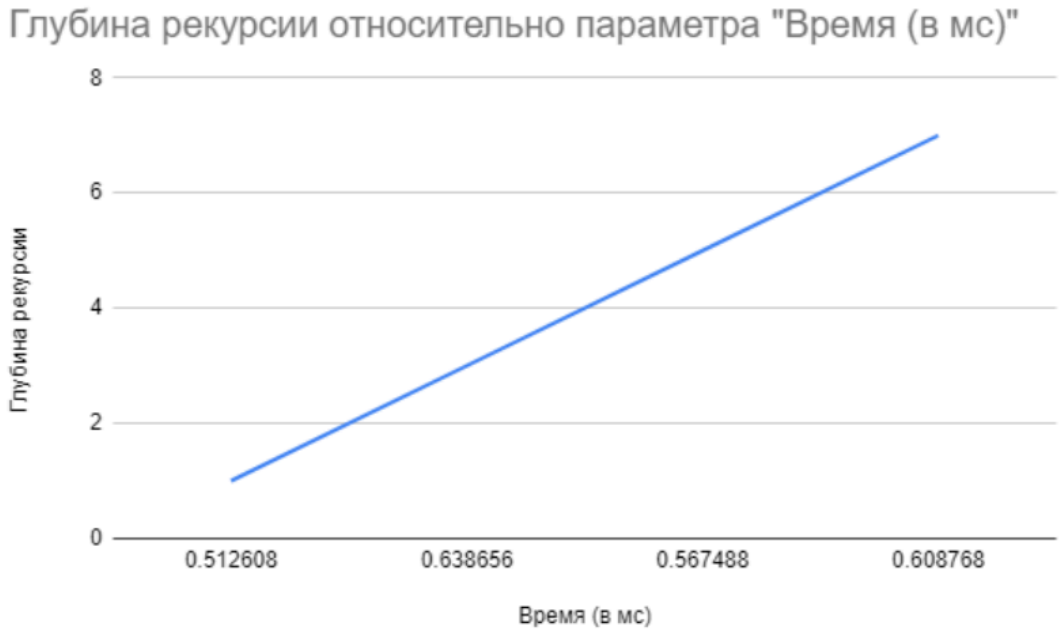
```
  if (ix_ < w_ && iy_ < h_) {
    Vector3 rDir(-1.0 + ix_ * dx_, -1.0 + iy_ * dy_, distPlane);
    rDir.yComp *= aspect;
    rDir = rDir.transformBy(rght, trueUp, fwd);

    ColorRGBA col_ = rayTrace(camLoc, rDir, 1e-5, bigVal,
      shapes, shapeCount, lamp, 5);

    double rr = col_.rChan, gg = col_.gChan, bb = col_.bChan;
    if (rr > 1) rr = 1; if (rr < 0) rr = 0;
    if (gg > 1) gg = 1; if (gg < 0) gg = 0;
    if (bb > 1) bb = 1; if (bb < 0) bb = 0;

    int idx_ = (h_ - 1 - iy_) * w_ + ix_;
    outFB[idx_].r = (unsigned char)(rr * 255);
    outFB[idx_].g = (unsigned char)(gg * 255);
    outFB[idx_].b = (unsigned char)(bb * 255);
    outFB[idx_].a = 255;
  }
}
```

Исследовательская часть и результаты

Замеры времени работы ядра с различными конфигурациями . Тесты будут для следующих размеров глубины рекурсии: 1, 3, 5, 7. А также будет приведён замер работы аналогичной программы без использования технологии CUDA - на CPU. Размер изображения 1024x768.

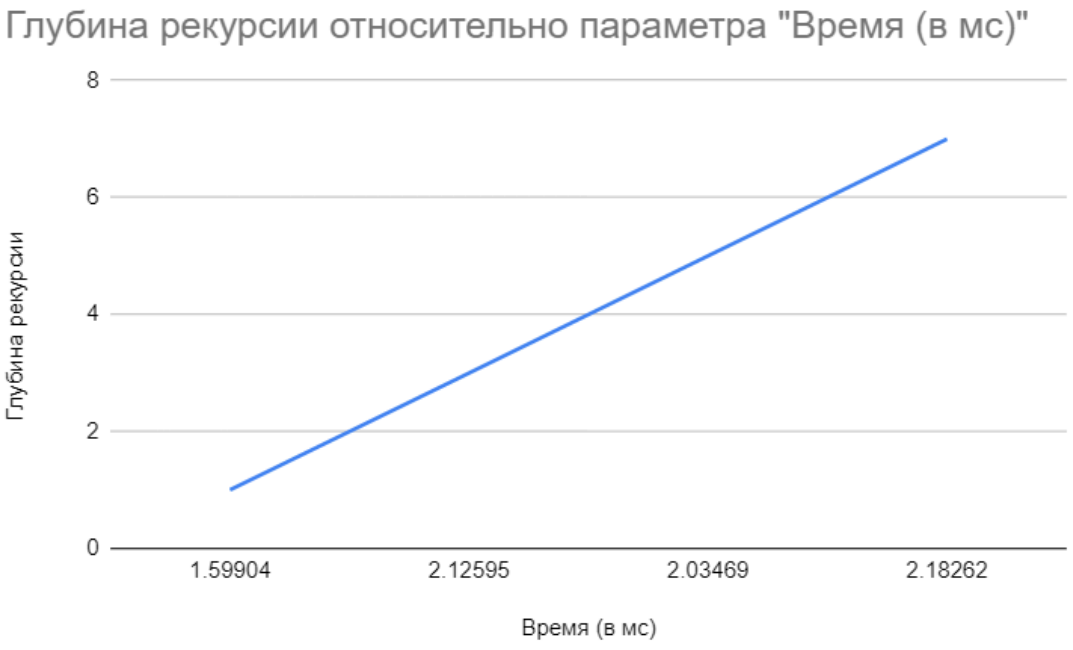
1. **Конфигурация <<< dim3(2, 2), dim3(4, 4) >>>**

Глубина рекурсии	Время (в мс)
1	0.512608
3	0.638656
5	0.567488
7	0.608768



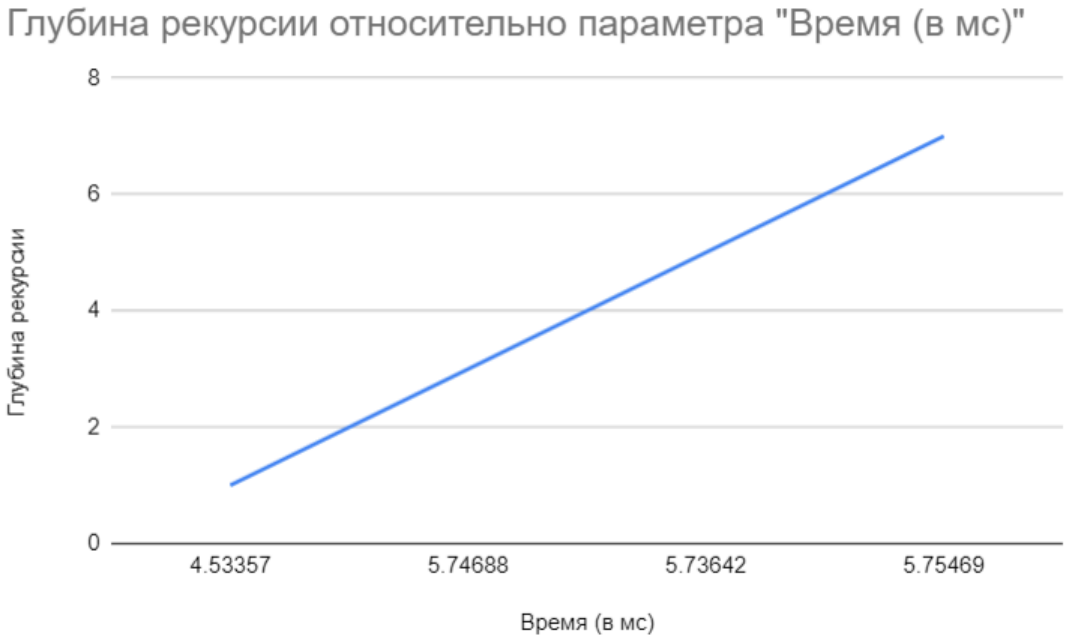
2. **Конфигурация <<< dim3(8, 8), dim3(16, 16) >>>**

Глубина рекурсии	Время (в мс)
1	1.59904
3	2.12595
5	2.03469
7	2.18262



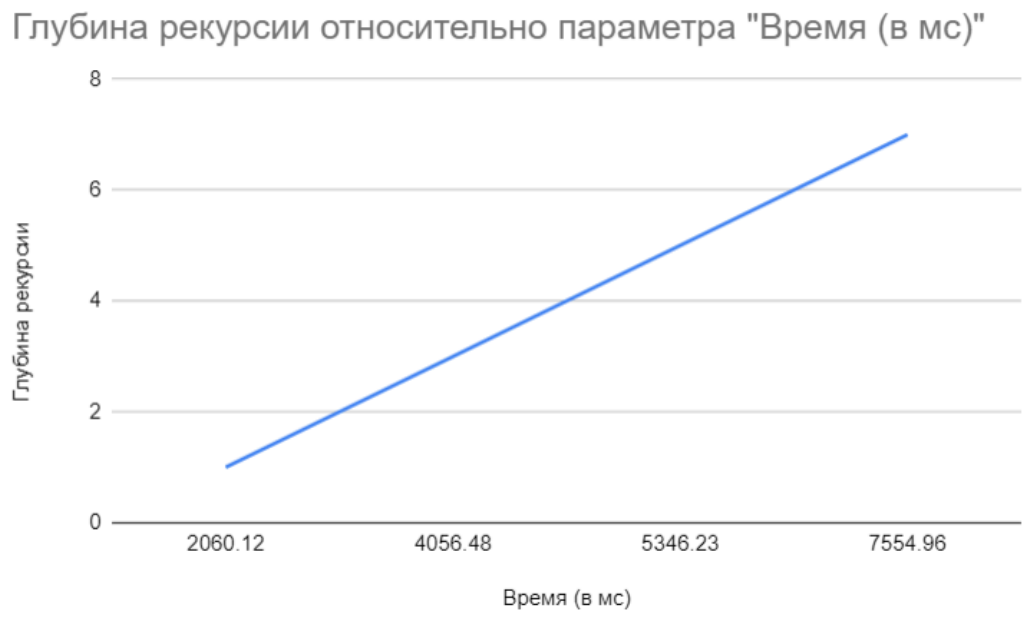
3. **Конфигурация <<< dim3(16, 16), dim3(16, 16) >>>**

Глубина рекурсии	Время (в мс)
1	4.53357
3	5.74688
5	5.73642
7	5.75469



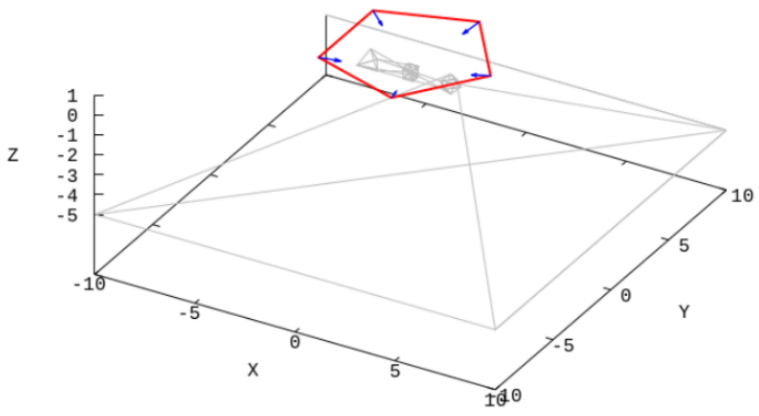
4. **CPU**

Глубина рекурсии	Время (в мс)
1	2060.12
3	4056.48
5	5346.23
7	7554.96

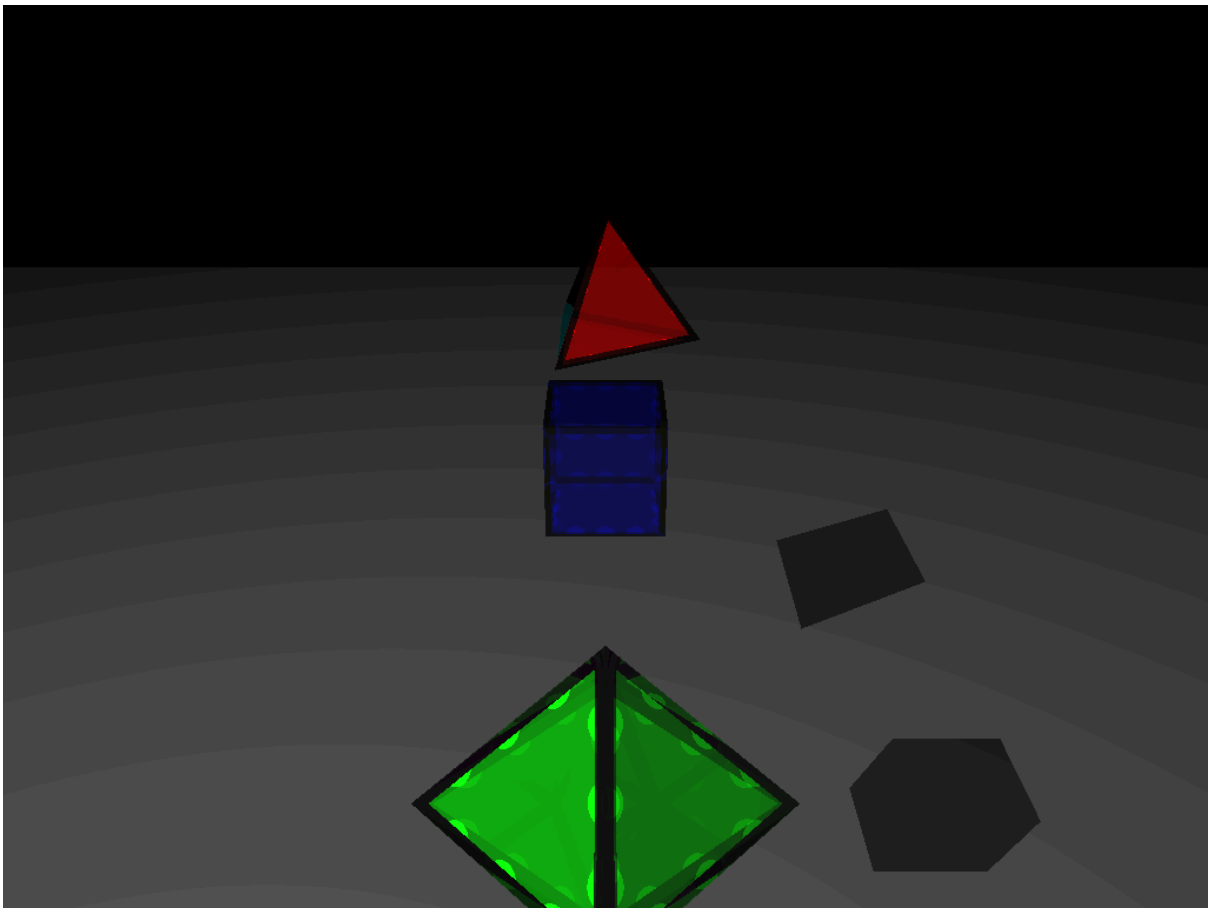


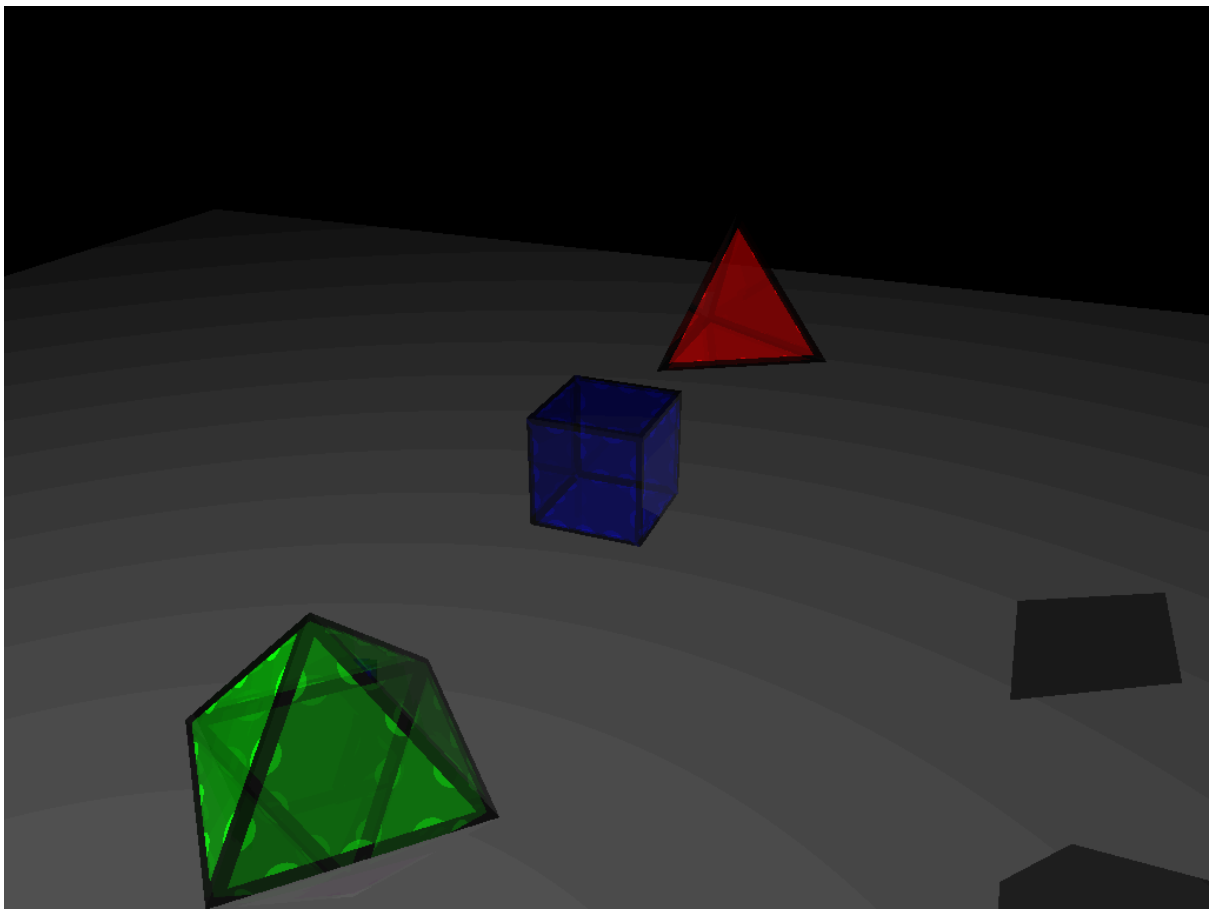
Входные данные на которых получается наиболее красочный результат:
600 1024 768 60 4.0
0 -3 1
0 0 0
3 -3 5
1 1 1
-2 0 0 1.0
0 0 0 1.0
2 0 0 1.0

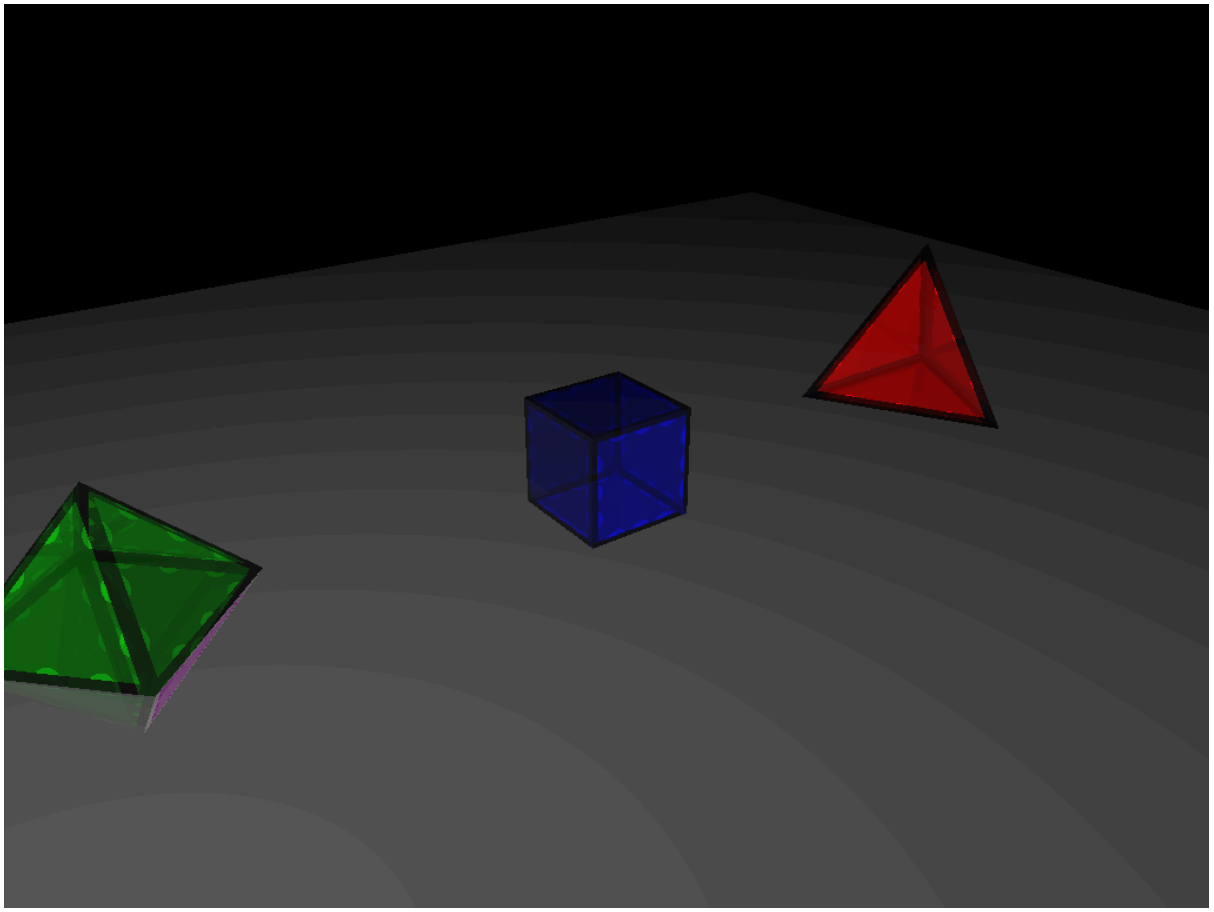
Трехмерных графиков (построен с помощью gnuplot), содержащий все полигоны сцены, траекторию облета камеры (красная линия) и траекторию направления камеры (синие вектора):

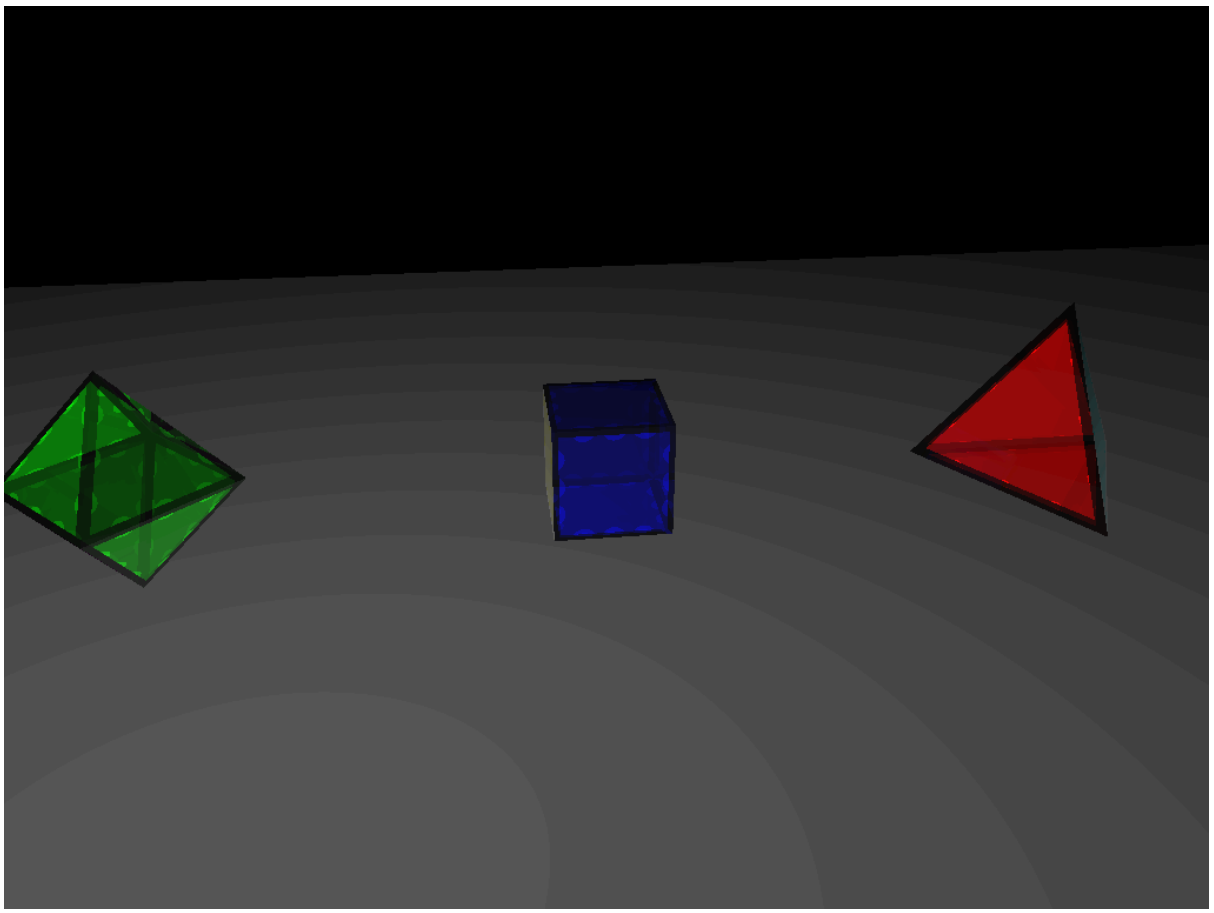


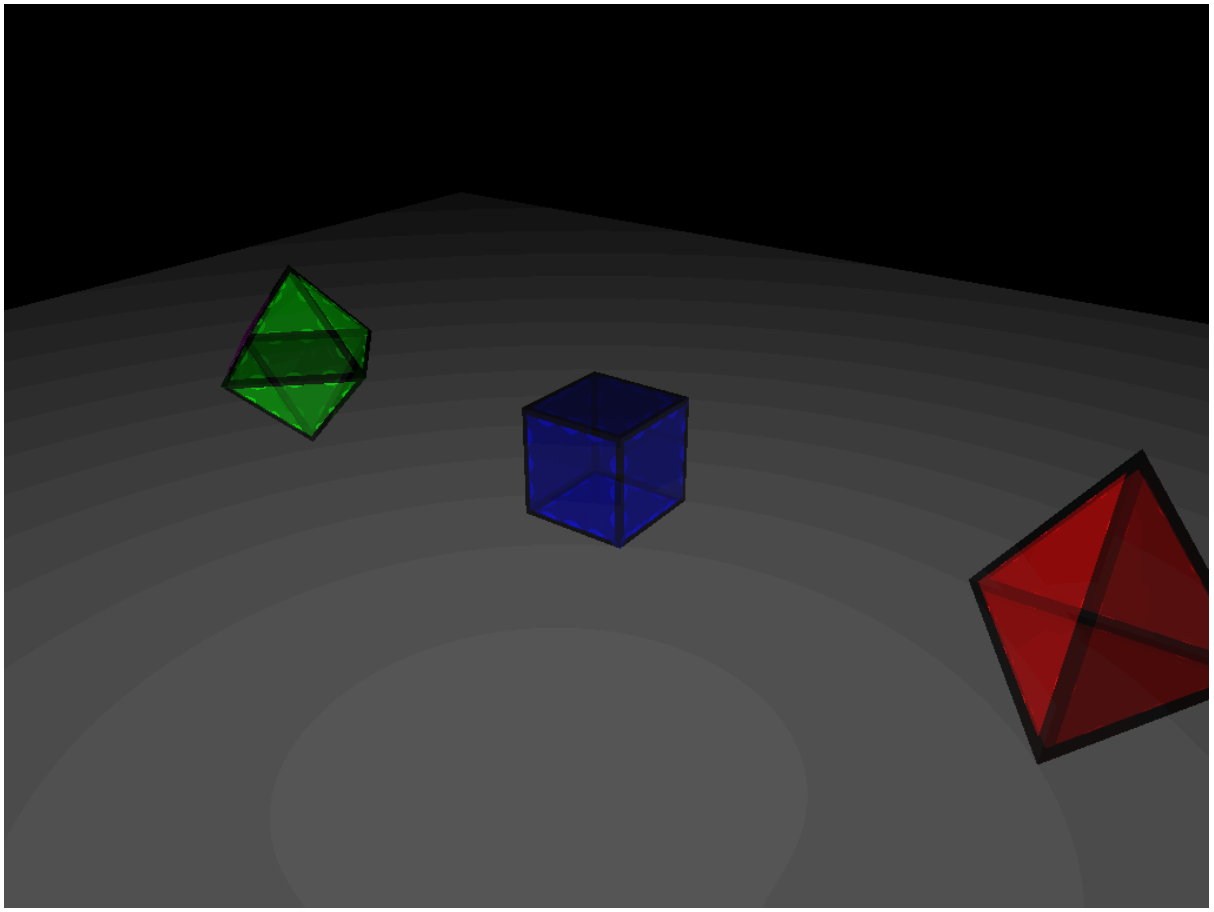
Различные кадры облёта фигур вокруг них:

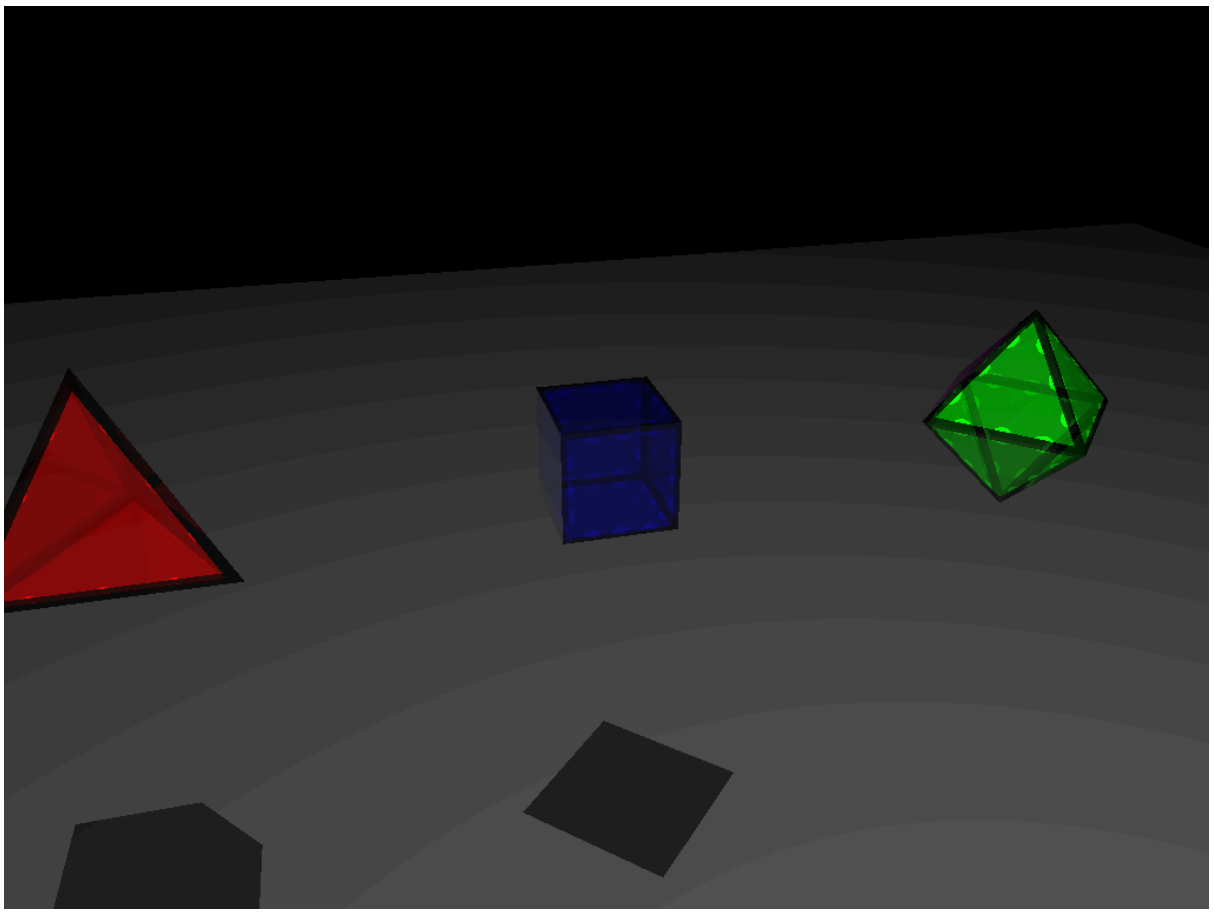


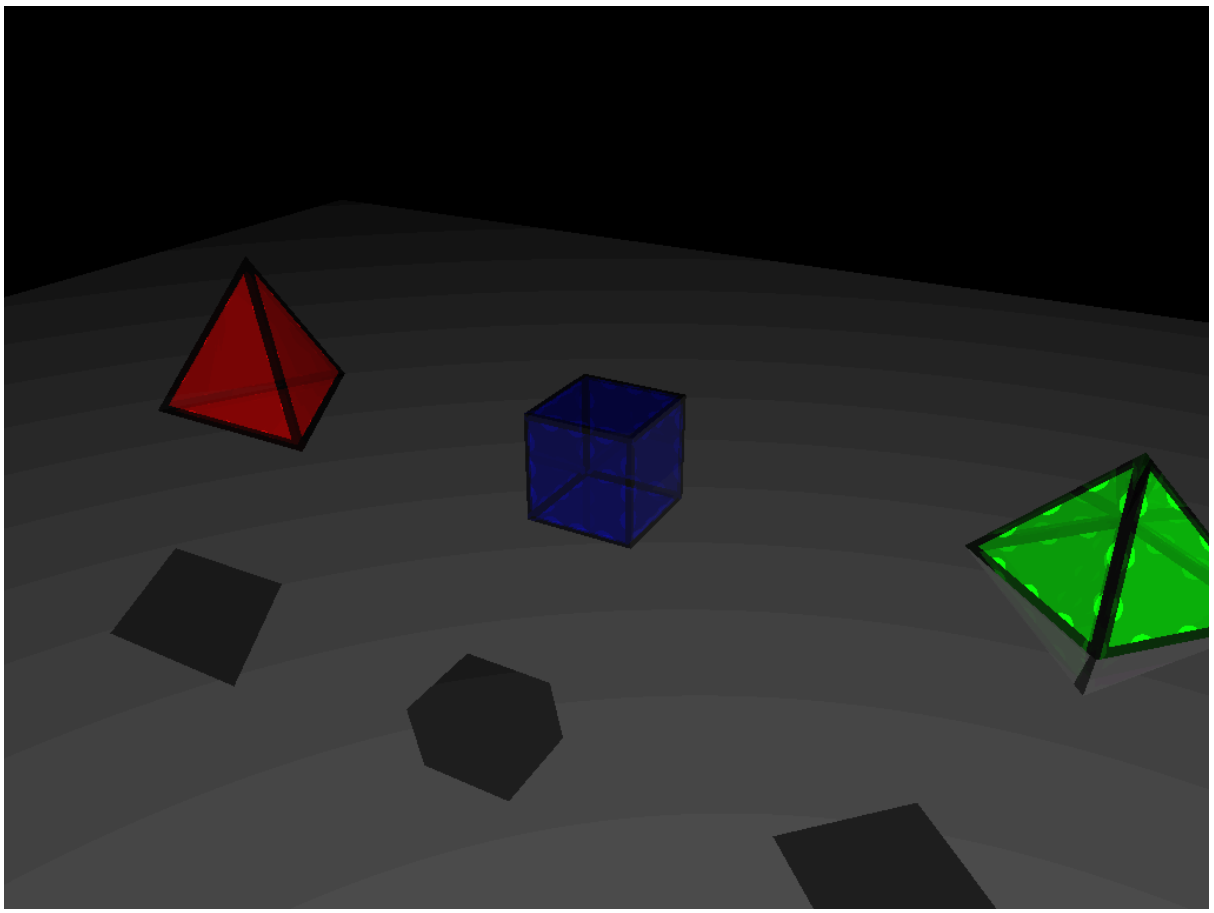


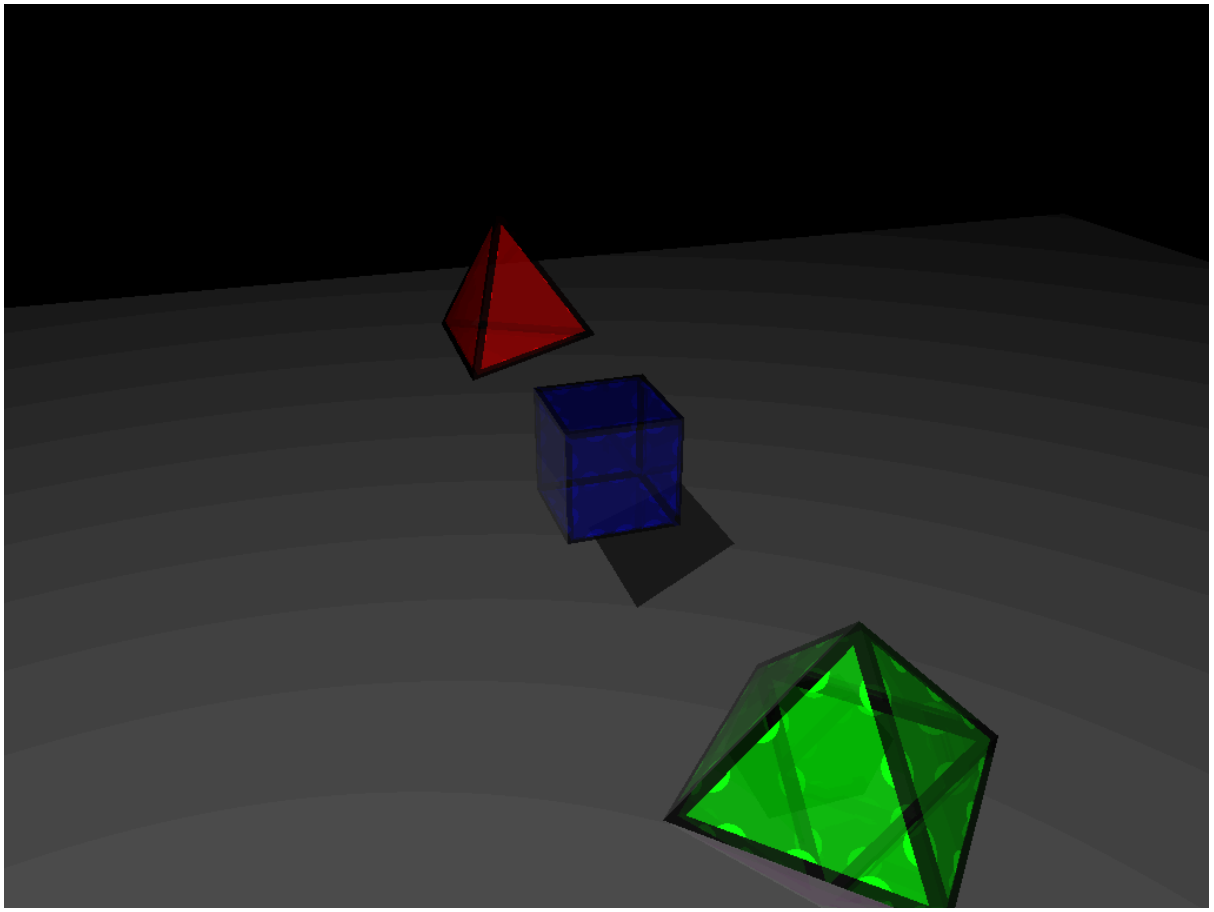


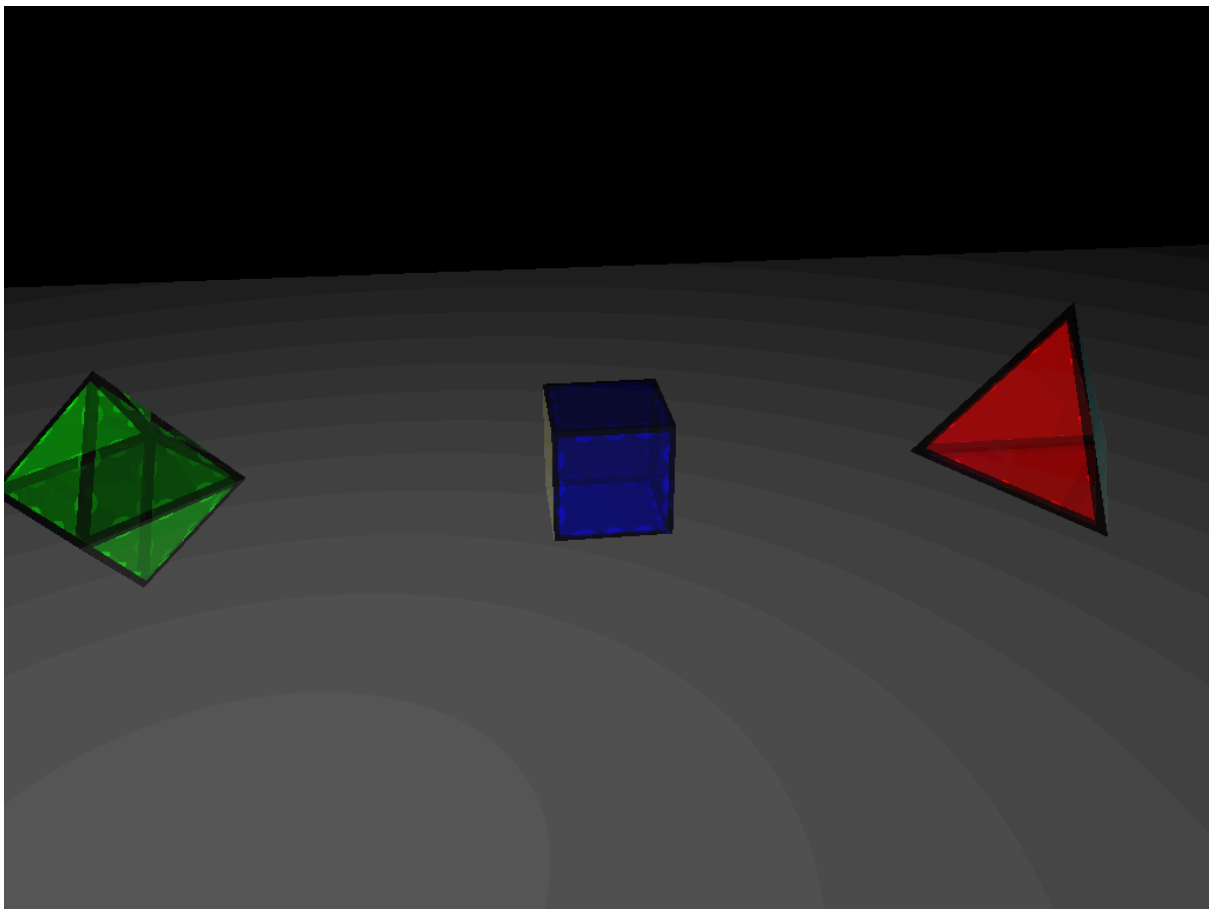


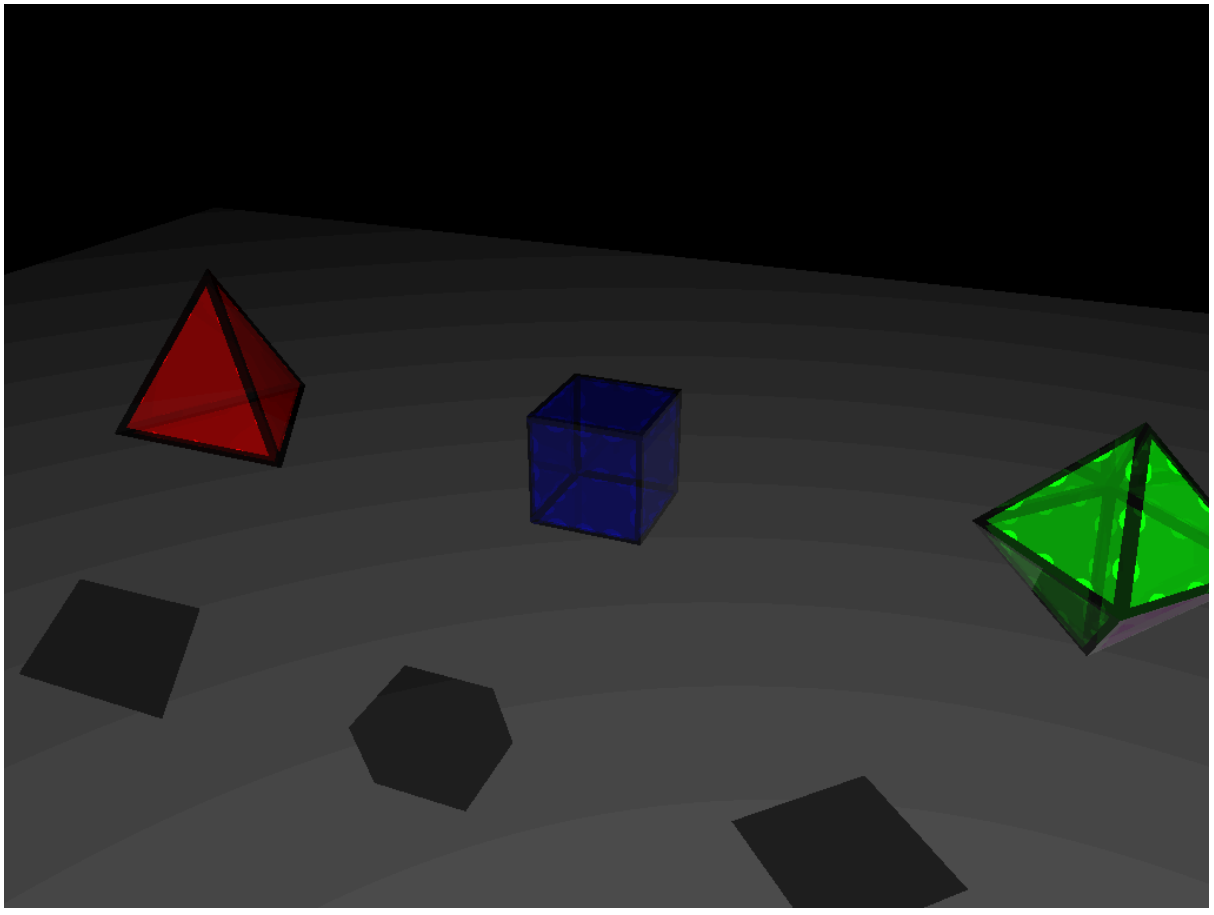












Выводы

В ходе выполнения курсовой работы я научился использовать GPU для создания фотореалистичной визуализации. Реализовал рендеринг полузеркальных и полупрозрачных правильных геометрических тел: гексаэдр, тетраэдр и октаэдр. Создал анимацию.

Этот алгоритм (рекурсивная трассировка лучей + модели освещения) может применяться в компьютерной графике и анимации (создание реалистичных кадров 3Д моделей, в видеоиграх для более точных освещения, теней и отражений), в области архитектурной визуализации (визуализация интерьеров и экстерьеров зданий с реалистичным освещением), в дизайне и моделированнии в промышленности (симуляция поведения материалов при различных условиях освещения) и в других областях.

В процессе написания программы я столкнулся с некоторыми сложностями:
1) Со сложностью реализации чёрной обводки у гексаэдра (куба), потому что я делал это с помощью добавления чёрной обводки тех треугольников, что формируют объекты, но это создавало чёрные полосы на гранях куба (диагонали квадратов, что его формируют), а также на полу (так как это тоже объединённые 2 треугольника), пришлось делать проверку на такое соединение треугольников и оставлять исходный цвет в этом случае.
2) Ещё одной сложностью стала реализация огоньков на всех гранях, пришлось подумать как их реализовать, до красивого варианта, как у крутых работ, у меня дойти не получилось. У меня просто вокруг этих “мини-светильников” становятся пиксели в два (можно другое задать число) раза ярче, из-за чего образовываются такие кругляшки на гранях фигур.
3) Также был момент с неправильным Ламбертовым освещением и затенением Фонга (очень блёклым всё получалось), но я поправил
4) Зеркальный эффект у объектов у меня не создаёт красивого эффекта бесконечности (можно сделать зеркальный эффект побольше, но тогда просто всё становится не очень красивым)

Сравнивая время работы ядра при рендеринге на GPU с различными конфигурациями и глубиной рекурсии, а также аналогичной программы без использования технологии CUDA - на CPU, можно заметить, что программа на CPU всегда проигрывает даже самой маленькой конфигурации на GPU. При этом все конфигурации и реализация на CPU растут линейно, однако интересно то, что чем больше была конфигурация, тем больше времени требовалось на создание одного кадра. Возможно, если бы картинка была более сложной и большой, а глубина рекурсии ещё больше, то это бы поменялось. Подведя итог, можно сказать, что все конфигурации на GPU всегда выигрывают по скорости у CPU, причём значительно, поэтому есть смысл применять при рендеринге именно GPU.

Литература

1) 3d движок, трассировка лучей в реальном времени, интерактивная трассировка URL: http://www.ray-tracing.ru/