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: А. Ю. Голов

Группа: 8О-401Б-21

Преподаватель: А.Ю. Морозов

Москва, 2025

**Условие**

Использование GPU для создание фотореалистической визуализации.

Рендеринг полузеркальных и полупрозрачных правильных геометрических тел.

Получение эффекта бесконечности. Создание анимации. 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

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). На сцене должны располагаться три тела: Тетраэдр, Гексаэдр, Икосаэдр.

**Программное и аппаратное обеспечение**

Видеокарта: NVIDIA GeForce RTX 2060 Mobile

1. **Compute Capability**: 7.5
2. **Графическая память**: 6 ГБ GDDR6, с 192-битной шиной и пропускной способностью 336 ГБ/с
3. **Разделяемая память**: до 64 КБ на мультипроцессор
4. **Константная память**: 64 КБ
5. **Количество регистров на блок**: 65 536
6. **Максимальное количество блоков на мультипроцессор**: 16
7. **Максимальное количество нитей на мультипроцессор**: 1 024
8. **Количество мультипроцессоров (SM)**: 30

Процессор: **Intel Core i7-9750H** имеет следующие характеристики:

* **Количество ядер**: 6
* **Количество потоков**: 12
* **Техпроцесс**: 14 нм

Оперативная память:

* Объём: 16 ГБ
* **Тактовая частота:** 3500 MHz
* **Поколение**: DDR4

Жёсткий диск:

* **Объём: 512 ГБ**
* Формат: SSD M2

Программное обеспечение:

* **Операционная система: Ubuntu 24.04 LTS**
* IDE: Lunar Vim

**Метод решения**

Фигуры и пол, представленные на сцене описываются классическими треугольными полигонами, задаваемыми тремя точками в пространстве и цветом. Количество полигонов определяется динамически, следовательно, количество объектов не ограничивается как-либо константным количеством полигонов. Для реализации эффекта отражения используется трассировка лучей с глубиной рекурсии 4, причём отражённый луч имеет как характеристику цвета, так и коэффициент отражаемости. Для сглаживания используется алгоритм SSAA — обрабатывающий кадр в более высоком разрешении для более высокого качества результирующих данных. Время, требуемое для вычислений, замеряется при вычислениях на видеокарте с помощью cudaEvent, и с помощью chrono при вычислениях на центральном процессоре.

**Описание программы**

Структура проекта выглядит следующим образом:

run.sh

conv.py

main.cu

Makefile

src/

….camera.cuh

….cpu.cuh

….figures.cuh

….floor.cuh

….gpu.cuh

….params.cuh

….point.cuh

….polygon.cuh

….ray.cuh

….scene.cuh

….transParams.cuh

Рассмотрим каждый из файлов проекта в порядке введения их содержимого в программный код.

**src/params.cuh**

В данном файле реализованы классы для хранения входных параметров, префикс каждого класса — P, поскольку в этих классах не реализован какой-либо функционал и все структуры являются датаклассами. Так же в этом файле для каждого класса перегружен оператор ввода во избежание излишней нагрузки на датаклассы и простоты использования реализуемых структур. Приведу пример одной структуры, аналогичной всем упомнятутым.

class PFrame

{

public:

int amount;

std::string path;

int width, height;

double angle;

};

std::istream& operator>>(std::istream& fin, PFrame& data)

{

fin >> data.amount >> data.path >> data.width >> data.height >> data.angle;

return fin;

}

**src/point.cuh**

В данном файле реализован класс Tpoint, для которого использован префикс — T, поскольку класс имеет собственные методы. Для безопасности данных все обращения к координатам и цвету реализуются при помощи геттеров и сеттеров. Так же для класса определены скалярное, матричное и векторное произведения.

class TPoint

{

public:

\_\_host\_\_ \_\_device\_\_ TPoint() {}

\_\_host\_\_ \_\_device\_\_ TPoint(double xVal, double yVal, double zVal)

{

x = xVal;

y = yVal;

z = zVal;

}

\_\_host\_\_ \_\_device\_\_ double GetX(){

return x;

}

\_\_host\_\_ \_\_device\_\_ double GetY(){

return y;

}

\_\_host\_\_ \_\_device\_\_ double GetZ(){

return z;

}

\_\_host\_\_ \_\_device\_\_ void SetX(double val){

x = val;

}

\_\_host\_\_ \_\_device\_\_ void SetY(double val){

y = val;

}

\_\_host\_\_ \_\_device\_\_ void SetZ(double val){

z = val;

}

\_\_host\_\_ \_\_device\_\_ TPoint operator+(TPoint p) {

return TPoint(x + p.x, y + p.y, z + p.z);

}

\_\_host\_\_ \_\_device\_\_ TPoint operator-(TPoint p) {

return TPoint(x - p.x, y - p.y, z - p.z);

}

\_\_host\_\_ \_\_device\_\_ TPoint operator\*(double num) {

return TPoint(x \* num, y \* num, z \* num);

}

\_\_host\_\_ \_\_device\_\_ TPoint normalize()

{

double l = sqrt(ScalarProd(\*this, \*this));

return TPoint(x / l, y / l, z / l);

}

friend std::istream& operator>>(std::istream& fin, TPoint& p);

friend std::ostream& operator<<(std::ostream& fout, TPoint& p);

friend \_\_host\_\_ \_\_device\_\_ double ScalarProd(TPoint p1, TPoint p2);

private:

double x, y, z;

};

std::istream& operator>>(std::istream& fin, TPoint& p) {

fin >> p.x >> p.y >> p.z;

return fin;

}

std::ostream& operator<<(std::ostream& fout, TPoint& p) {

fout << p.x << " " << p.y << " " << p.z;

return fout;

}

\_\_host\_\_ \_\_device\_\_ double ScalarProd(TPoint p1, TPoint p2) {

return p1.GetX() \* p2.GetX() + p1.GetY() \* p2.GetY() + p1.GetZ() \* p2.GetZ();

}

\_\_host\_\_ \_\_device\_\_ TPoint VectorProd(TPoint p1, TPoint p2) {

return TPoint(p1.GetY() \* p2.GetZ() - p1.GetZ() \* p2.GetY(),

p1.GetZ() \* p2.GetX() - p1.GetX() \* p2.GetZ(),

p1.GetX() \* p2.GetY() - p1.GetY() \* p2.GetX());

}

\_\_host\_\_ \_\_device\_\_ TPoint MatrixProd(TPoint p1, TPoint p2, TPoint p3, TPoint p4) {

return TPoint(p1.GetX() \* p4.GetX() + p2.GetX() \* p4.GetY() + p3.GetX() \* p4.GetZ(),

p1.GetY() \* p4.GetX() + p2.GetY() \* p4.GetY() + p3.GetY() \* p4.GetZ(),

p1.GetZ() \* p4.GetX() + p2.GetZ() \* p4.GetY() + p3.GetZ() \* p4.GetZ());

}

**src/polygon.cuh**

В данном файле реализован класс Tpolygon, представляющий из себя три экземпляра класса Tpoint — треугольник и цвет. Работа с данными так же реализована с помощью геттеров и сеттеров.

**src/floor.cuh**

В данном файле реализован абстрактный класс Ifloor, декларирующий структуру класса Tfloor, суть которого заключается в добавлении в пространство пола прямоугольной формы.

class IFloor

{

public:

virtual void AddFigure(std::vector<TPolygon> &canvas) = 0;

virtual ~IFloor() {};

protected:

PFloor params;

};

class TFloor : public IFloor

{

public:

TFloor() {};

TFloor(PFloor paramsValues){

params = paramsValues;

}

void AddFigure(std::vector<TPolygon> &canvas) override

{

canvas.emplace\_back(TPolygon(params.p1, params.p2, params.p3, params.colour));

canvas.emplace\_back(TPolygon(params.p1, params.p3, params.p4, params.colour));

}

};

**src/figures.cuh**

В данном файле по аналогии с IFloor и TFloor реализованы заданные стереометрические фигуры. Для краткости приведу реализацию гексаэдра.

class IFigure

{

public:

virtual void AddFigure(std::vector<TPolygon>& canvas) = 0;

virtual ~IFigure() = default;

protected:

PFigure params;

};  
  
class THexahedron : public IFigure

{

public:

explicit THexahedron(PFigure paramsValues) { params = paramsValues; }

void AddFigure(std::vector<TPolygon>& canvas) override

{

double a = params.radius \* 2;

std::vector<TPoint> v = {

{params.center.GetX() - a / 2, params.center.GetY() - a / 2, params.center.GetZ() - a / 2},

{params.center.GetX() + a / 2, params.center.GetY() - a / 2, params.center.GetZ() - a / 2},

{params.center.GetX() + a / 2, params.center.GetY() + a / 2, params.center.GetZ() - a / 2},

{params.center.GetX() - a / 2, params.center.GetY() + a / 2, params.center.GetZ() - a / 2},

{params.center.GetX() - a / 2, params.center.GetY() - a / 2, params.center.GetZ() + a / 2},

{params.center.GetX() + a / 2, params.center.GetY() - a / 2, params.center.GetZ() + a / 2},

{params.center.GetX() + a / 2, params.center.GetY() + a / 2, params.center.GetZ() + a / 2},

{params.center.GetX() - a / 2, params.center.GetY() + a / 2, params.center.GetZ() + a / 2}

};

int faces[6][4] = {

{0, 1, 2, 3}, {4, 5, 6, 7}, {0, 1, 5, 4},

{2, 3, 7, 6}, {1, 2, 6, 5}, {0, 3, 7, 4}

};

for (const auto& face : faces)

{

canvas.emplace\_back(v[face[0]], v[face[1]], v[face[2]], params.colour);

canvas.emplace\_back(v[face[0]], v[face[2]], v[face[3]], params.colour);

}

}

};

**src/transParams.cuh**

В данном файле реализованы датаклассы, используемые для передачи декларированного набора параметров в процедуры, подразумевающие вычисления на графическом процессоре.

struct TRayParams

{

int width;

int height;

double angle;

TPoint lightPos;

uchar4 lightColour;

TPolygon\* canvas;

int canvasSize;

TRayParams(int widthVal, int heightVal, double angleVal, TPoint lightPosVal, uchar4 lightColourVal, TPolygon\* canvasVal, int canvasSizeVal)

{

width = widthVal;

height = heightVal;

angle = angleVal;

lightPos = lightPosVal;

lightColour = lightColourVal;

canvas = canvasVal;

canvasSize = canvasSizeVal;

}

};

struct TSmoothParams

{

int width;

int height;

int rayPerPixel;

TSmoothParams(int widthVal, int heightVal, double rayPerPixelVal)

{

width = widthVal;

height = heightVal;

rayPerPixel = rayPerPixelVal;

}

};

**src/camera.cuh**

В этом файле реализован класс TСamera, хранящий входные параметры, и реализующий методы, вычисляющие радиальные параметры позиции и направления камеры.

class TCamera

{

public:

TCamera() = default;

TCamera(PCamera paramsVal){

params = paramsVal;

}

double GetPosRadialX(double time){

return (params.r0c + params.arc \* sin(params.wrc \* time + params.prc)) \* cos(params.phi0c + params.wphic \* time);

}

double GetPosRadialY(double time){

return (params.r0c + params.arc \* sin(params.wrc \* time + params.prc)) \* sin(params.phi0c + params.wphic \* time);

}

double GetPosHeightZ(double time){

return params.z0c + params.azc \* sin(params.wzc \* time + params.pzc);

}

double GetDirRadialX(double time){

return (params.r0n + params.arn \* sin(params.wrn \* time + params.prn)) \* cos(params.phi0n + params.wphin \* time);

}

double GetDirRadialY(double time){

return (params.r0n + params.arn \* sin(params.wrn \* time + params.prn)) \* sin(params.phi0n + params.wphin \* time);

}

double GetDirHeightZ(double time){

return params.z0n + params.azn \* sin(params.wzn \* time + params.pzn);

}

private:

PCamera params;

};

**src/gpu.cuh и src/cpu.cuh**

В этих файлах представлены процедуры для рендеринга и сглаживания кадров на графическом и центральном процессорах соответственно. Приведу реализацию рендеринга и сглаживания на видеокарте.

\_\_global\_\_ void GPURender(int\* devRayCnt, uchar4\* pixelBuffer, TPoint cameraPosition, TPoint cameraDirection, TRayParams params)

{

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

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

int shiftX = blockDim.x \* gridDim.x;

int shiftY = blockDim.y \* gridDim.y;

double pixelWidth = 2.0 / (params.width - 1.0);

double pixelHeight = 2.0 / (params.height - 1.0);

double focalLength = 1.0 / tan(params.angle \* M\_PI / 360.0);

TPoint forwardVector = (cameraDirection - cameraPosition).normalize();

TPoint rightVector = VectorProd(forwardVector, {0.0, 0.0, 1.0}).normalize();

TPoint upVector = VectorProd(rightVector, forwardVector).normalize();

for (int x = posX; x < params.width; x += shiftX)

{

for (int y = posY; y < params.height; y += shiftY)

{

TPoint screenCoordinate = TPoint(-1.0 + pixelWidth \* x, (-1.0 + pixelHeight \* y) \* params.height / params.width, focalLength);

TPoint rayDirection = MatrixProd(rightVector, upVector, forwardVector, screenCoordinate);

pixelBuffer[(params.height - 1 - y) \* params.width + x] = TraceRay(cameraPosition, rayDirection.normalize(), params);

\*devRayCnt += 4;

}

}

}

\_\_global\_\_ void GPUSmoothing(uchar4\* inputBuffer, uchar4\* outputBuffer, TSmoothParams params)

{

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

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

int shiftX = blockDim.x \* gridDim.x;

int shiftY = blockDim.y \* gridDim.y;

for (int x = posX; x < params.width; x += shiftX)

{

for (int y = posY; y < params.height; y += shiftY)

{

uint4 colorAccumulator = make\_uint4(0, 0, 0, 0);

for (int i = 0; i < params.rayPerPixel; ++i)

{

for (int j = 0; j < params.rayPerPixel; ++j)

{

uchar4 currentPixel = inputBuffer[(params.width \* params.rayPerPixel \* (y \* params.rayPerPixel + j) + (x \* params.rayPerPixel + i))];

colorAccumulator.x += currentPixel.x;

colorAccumulator.y += currentPixel.y;

colorAccumulator.z += currentPixel.z;

}

}

int totalSamples = params.rayPerPixel \* params.rayPerPixel;

outputBuffer[y \* params.width + x] = make\_uchar4(colorAccumulator.x / totalSamples, colorAccumulator.y / totalSamples, colorAccumulator.z / totalSamples, 255);

}

}

}

**src/ray.cuh**

Файл содержит процедуру трассировки лучей с рекурсивным вычислением. Функция вычисляет луч с учётом его цвета и коэффициента отражения.

\_\_host\_\_ \_\_device\_\_ uchar4 TraceRay(TPoint position, TPoint direction, TRayParams params, int depth = 3)

{

int closestPolygonIndex = -1;

double closestIntersection;

for (int i = 0; i < params.canvasSize; ++i)

{

TPoint edge1 = params.canvas[i].GetP2() - params.canvas[i].GetP1();

TPoint edge2 = params.canvas[i].GetP3() - params.canvas[i].GetP1();

TPoint crossProduct = VectorProd(direction, edge2);

double determinant = ScalarProd(crossProduct, edge1);

if (fabs(determinant) < 1e-10) {

continue;

}

TPoint translationVector = position - params.canvas[i].GetP1();

TPoint crossTranslation = VectorProd(translationVector, edge1);

double barycentricU = ScalarProd(crossProduct, translationVector) / determinant;

double barycentricV = ScalarProd(crossTranslation, direction) / determinant;

if ((barycentricU < 0.0 || barycentricU > 1.0) || (barycentricV < 0.0 || barycentricV + barycentricU > 1.0)) {

continue;

}

double intersectionDistance = ScalarProd(crossTranslation, edge2) / determinant;

if (intersectionDistance < 0.0) {

continue;

}

if (closestPolygonIndex == -1 || intersectionDistance < closestIntersection)

{

closestPolygonIndex = i;

closestIntersection = intersectionDistance;

}

}

if (closestPolygonIndex == -1) {

return make\_uchar4(0, 0, 0, 255);

}

TPoint intersectionPoint = direction \* closestIntersection + position;

TPoint edge1 = params.canvas[closestPolygonIndex].GetP2() - params.canvas[closestPolygonIndex].GetP1();

TPoint edge2 = params.canvas[closestPolygonIndex].GetP3() - params.canvas[closestPolygonIndex].GetP1();

TPoint normalVector = VectorProd(edge1, edge2).normalize();

TPoint reflectedDirection = direction - normalVector \* (2.0 \* ScalarProd(direction, normalVector));

TPoint lightDirection = (params.lightPos - intersectionPoint).normalize();

double diffuseIntensity = fmax(0.25, ScalarProd(normalVector, lightDirection));

uchar4 polygonColor = params.canvas[closestPolygonIndex].GetColour();

uchar4 directColor = make\_uchar4(

polygonColor.x \* params.lightColour.x \* diffuseIntensity,

polygonColor.y \* params.lightColour.y \* diffuseIntensity,

polygonColor.z \* params.lightColour.z \* diffuseIntensity,

255

);

double reflectionCoefficient = 0.5;

if (depth > 0 && reflectionCoefficient > 0.0)

{

uchar4 reflectedColor = TraceRay(intersectionPoint, reflectedDirection, params, depth - 1);

directColor.x = directColor.x \* (1 - reflectionCoefficient) + reflectedColor.x \* reflectionCoefficient;

directColor.y = directColor.y \* (1 - reflectionCoefficient) + reflectedColor.y \* reflectionCoefficient;

directColor.z = directColor.z \* (1 - reflectionCoefficient) + reflectedColor.z \* reflectionCoefficient;

}

return directColor;

}

**src/scene.cuh**

Файл содержит класс TScene, хранящий параметры сцены, производящий вычисления как на CPU, так и на GPU. Так же класс хранит экземпляр класса TCamera и сохраняет данные кадра в бинарном файле.

**main.cuh**

Файл содержит функцию main(), в которой создаются экземпляры классов TCamera и TScene, и производятся вычисления в соответствии с задаваемыми параметрами.

**conv,py** — конвертирует бинарные данные в изображения

**run.sh** — скрипт, компилирующий программу, формирующий файл входных данных, и запускающий программу. После завершения работы программы бинарные данные конвертируются в изображения при помощи conv.py, которые утилитой ffmpeg преобразуются в файл .mp4

**Makefile**

# checker

NVCC = /usr/local/cuda/bin/nvcc

MPIC++ = /usr/local/bin/mpic++

local

NVCC = nvcc

SRC\_DIR = src

TARGET = cp

SRCS = $(wildcard $(SRC\_DIR)/\*.cu)

OBJS = $(SRCS:.cu=.o)

INCLUDE\_DIRS = -I$(SRC\_DIR)

NVCC\_FLAGS = --std=c++11 -Wno-deprecated-gpu-targets

.PHONY: all clean

all: $(TARGET)

$(TARGET): $(OBJS)

$(NVCC) $(NVCC\_FLAGS) -o $@ $^ $(INCLUDE\_DIRS)

$(SRC\_DIR)/%.o: $(SRC\_DIR)/%.cu

$(NVCC) $(NVCC\_FLAGS) -c $< -o $@ $(INCLUDE\_DIRS)

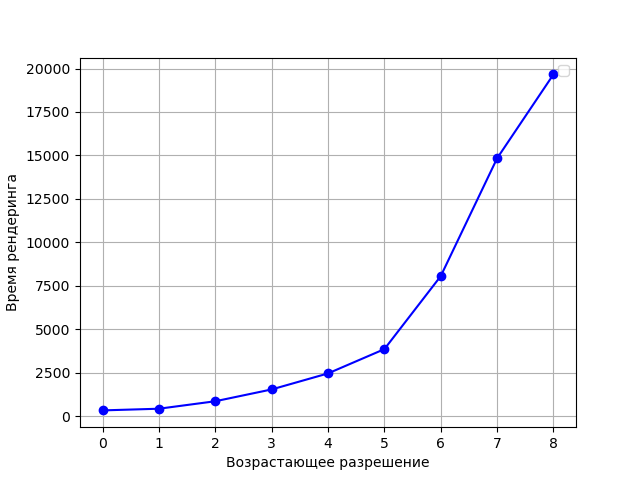
clean:

rm -f $(OBJS) $(TARGET)

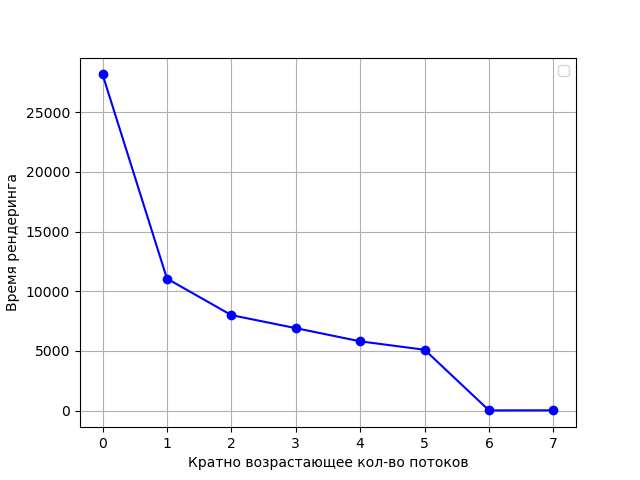
**Исследовательская часть и результаты**

Все измерения проводятся в миллисекундах.

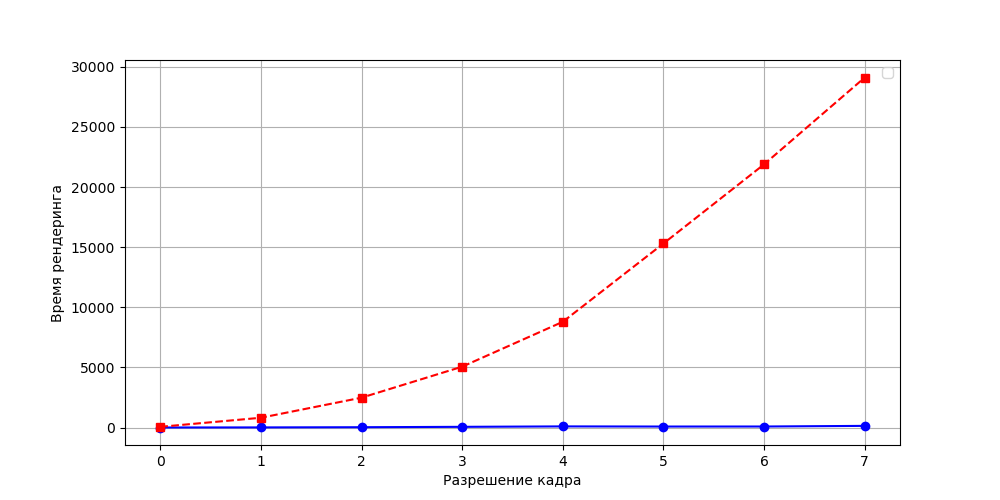
Проанализируем зависимости времени рендеринга кадра от его разрешения. Были взяты следующие пары чисел в качестве задаваемых размеров кадра: 640x480, 800x600, 1024x768, 1280x960, 1600x1200, 1920x1440, 2560x1440, 3200x1800, 3840x2160.  
Зависимость представлена на граифе, где по оси Y — возрастающее разрешение кадра. Зависимость очевидно квадратичная.



Теперь определим кадр в разрешении 4096x2160 пикселей. Варьируя конфигурации ядра, кратно увеличивая количество потоков, получим следующую зависимость:

****

Проведём сравнение производительности вычислений на GPU и CPU, для вычислений на центральном процессоре за разумное время будут взяты кадры с достаточно малым разрешением.

Для наглядности данных приведём те же данные в виде таблицы.

|  |  |  |
| --- | --- | --- |
| Кол-во кадров | GPU | CPU |
| 20 | 1.18522 | 58 |
| 80 | 8.30992 | 811 |
| 140 | 24.33 | 2482 |
| 200 | 56.6274 | 5061 |
| 260 | 92.5718 | 8795 |
| 320 | 80.9038 | 15309 |
| 380 | 83.6144 | 21896 |
| 440 | 136.37 | 29117 |

Приведём скриншоты результата при следующих параметрах:  
data-bin/%d.data

1920 1080 120

7 2 -0.5 1 1 1 1 1 0 0

2 0 0 0.5 0.1 1 1 1 0 0

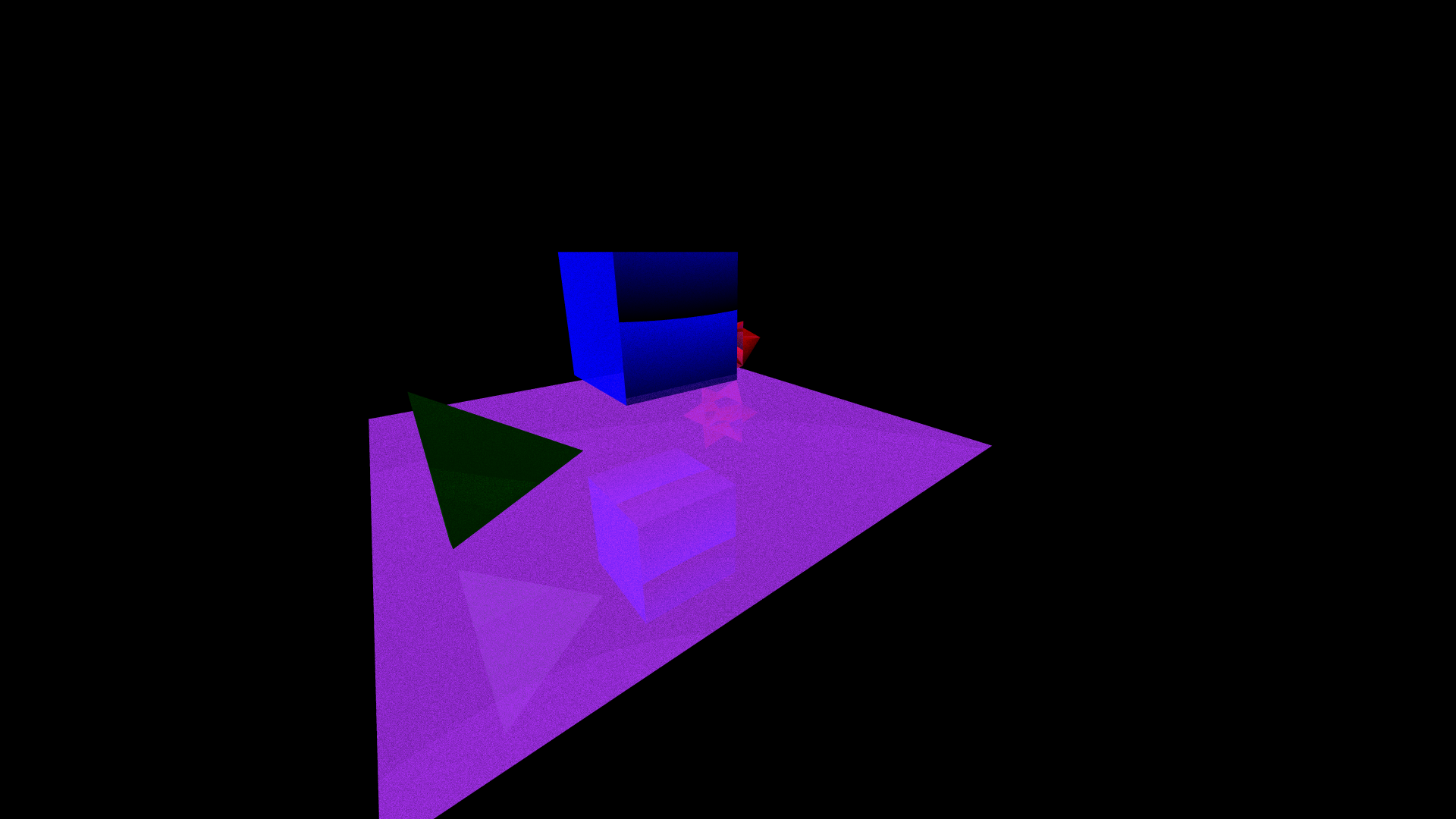
2 -3 0 0 1 0 1

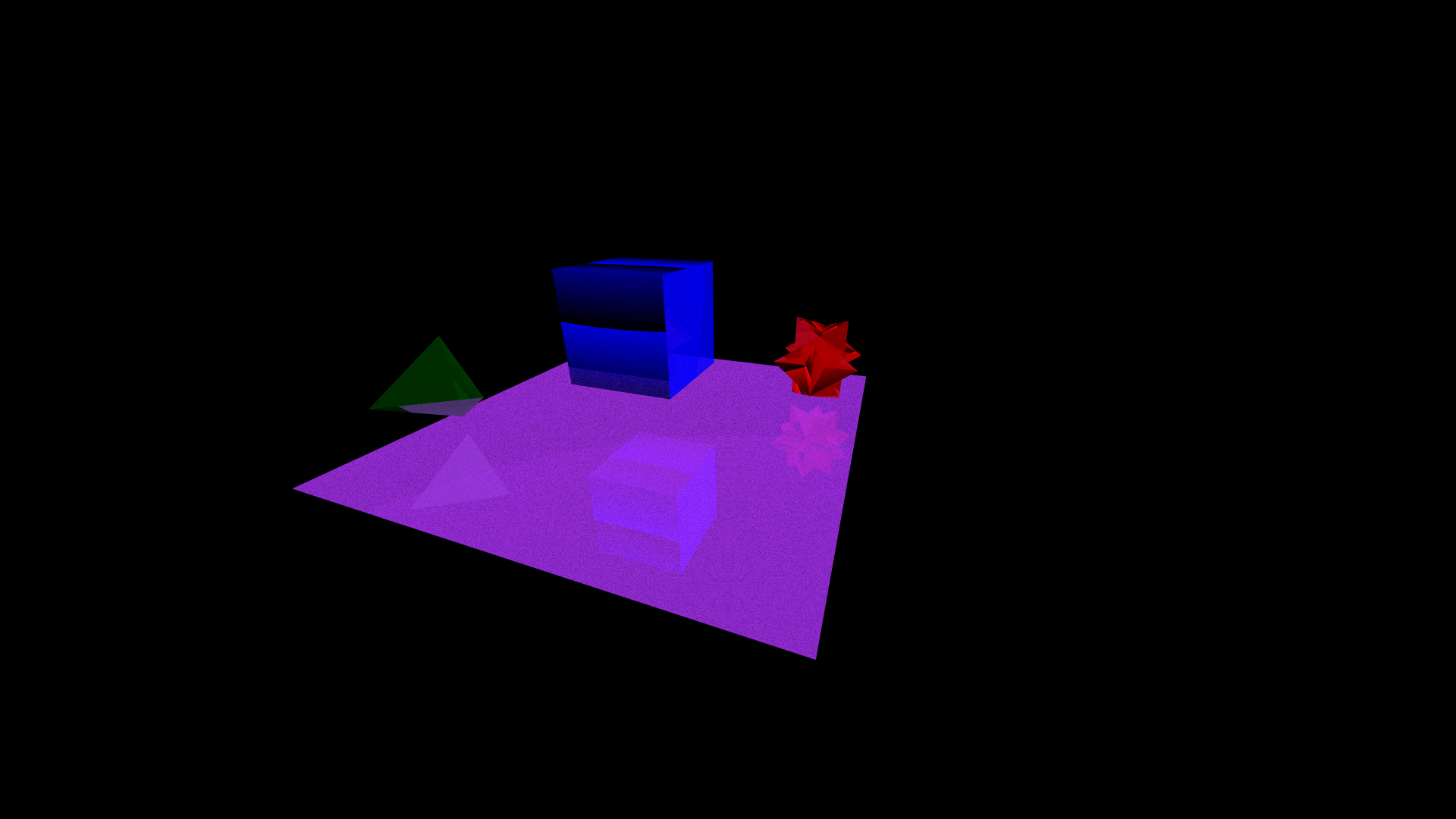
0 0 1 0 0 1 1

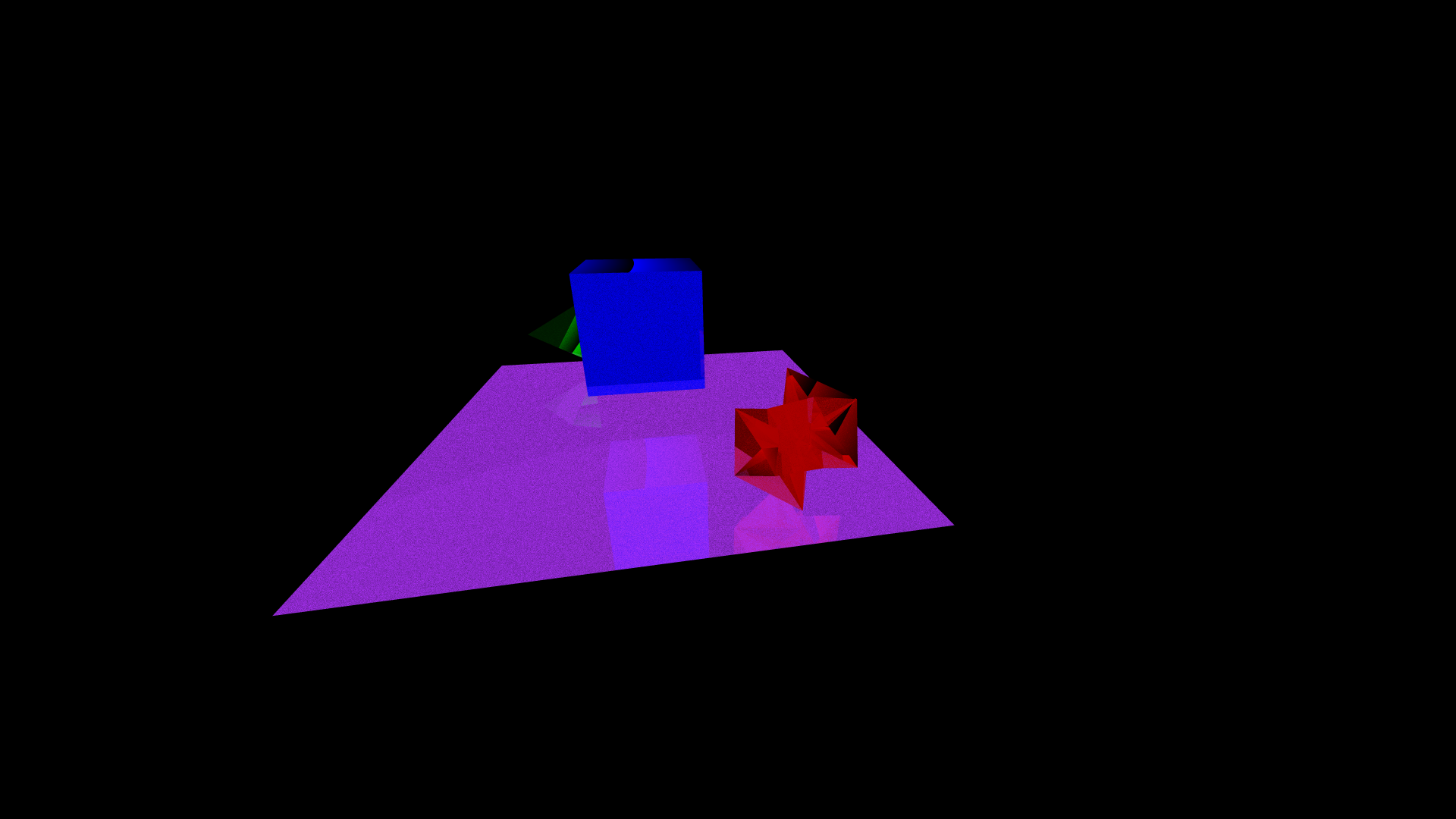
-2 3 0 1 0 0 1

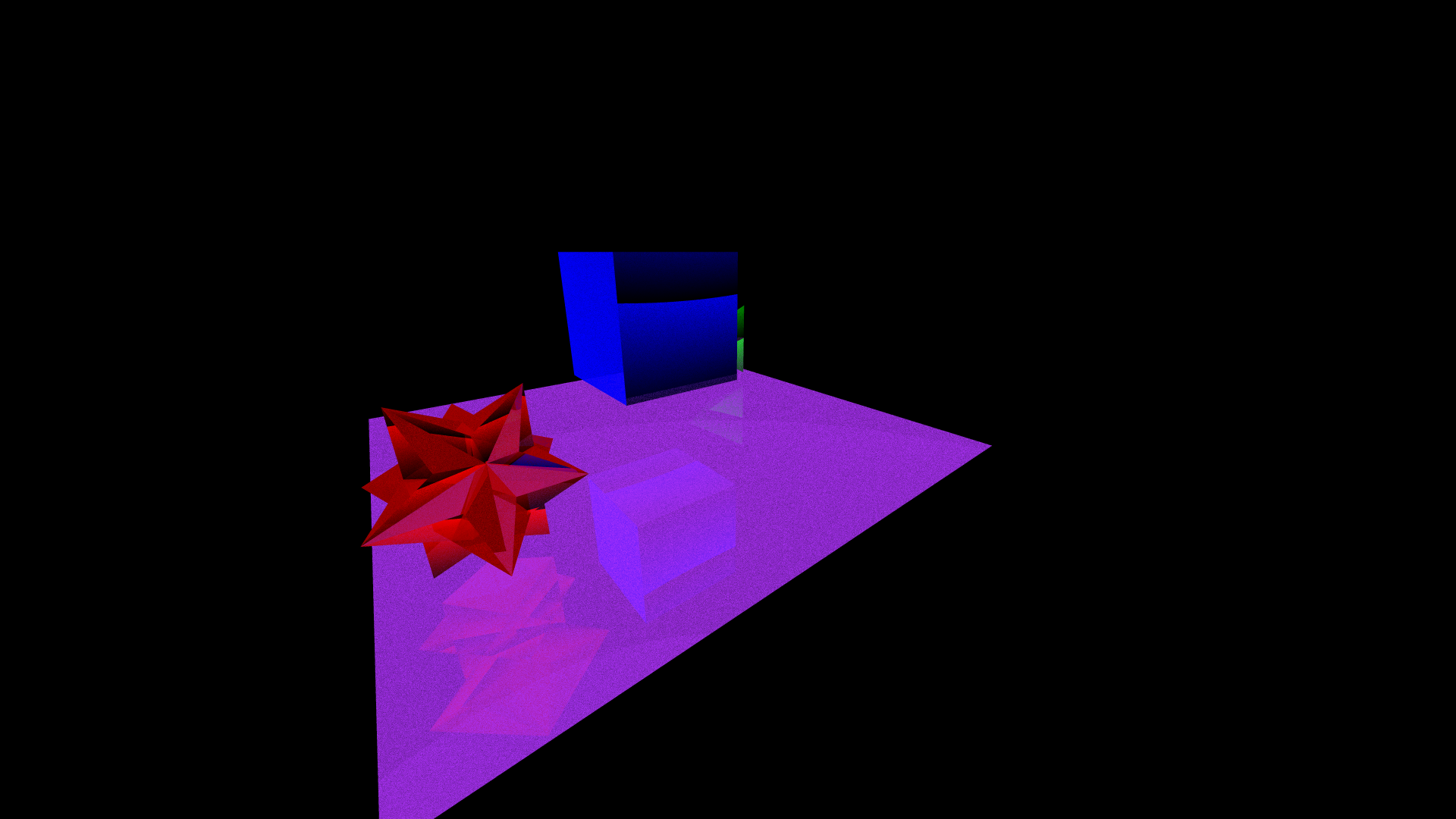
-4 -4 -1 -4 4 -1 4 4 -1 4 -4 -1 1 1 1

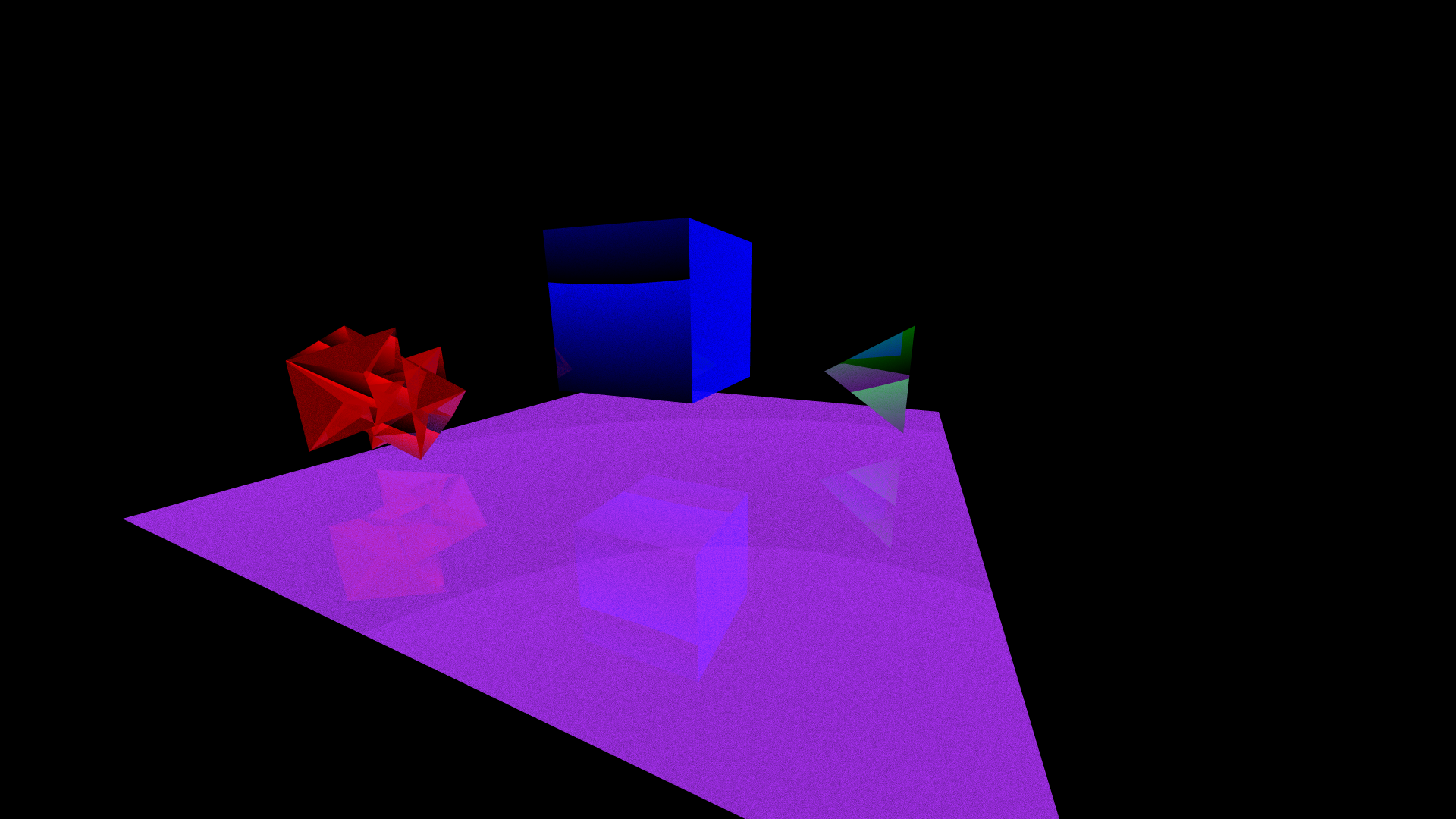
10 0 15 0.294118 0.196078 0.0980392 4

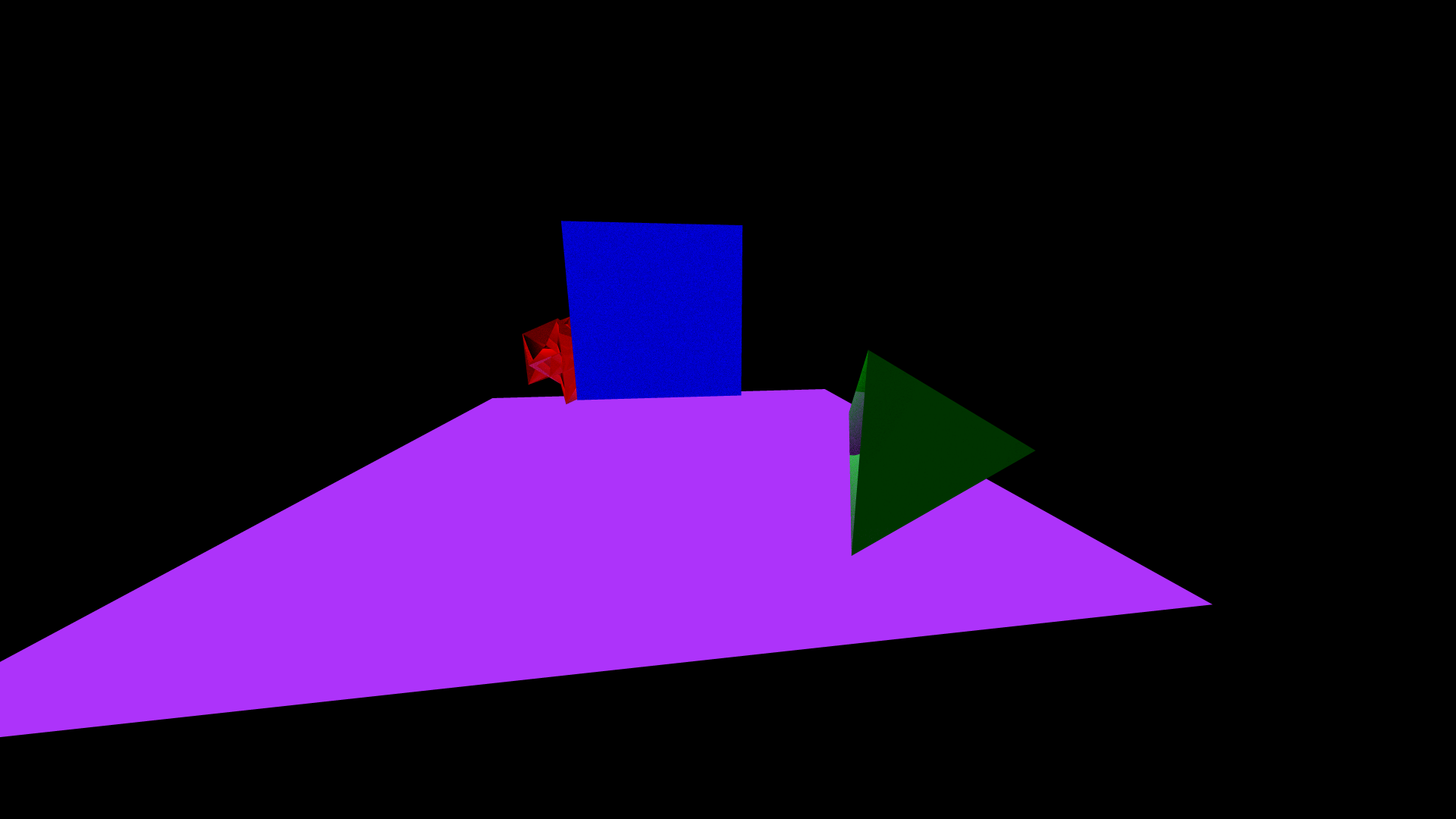


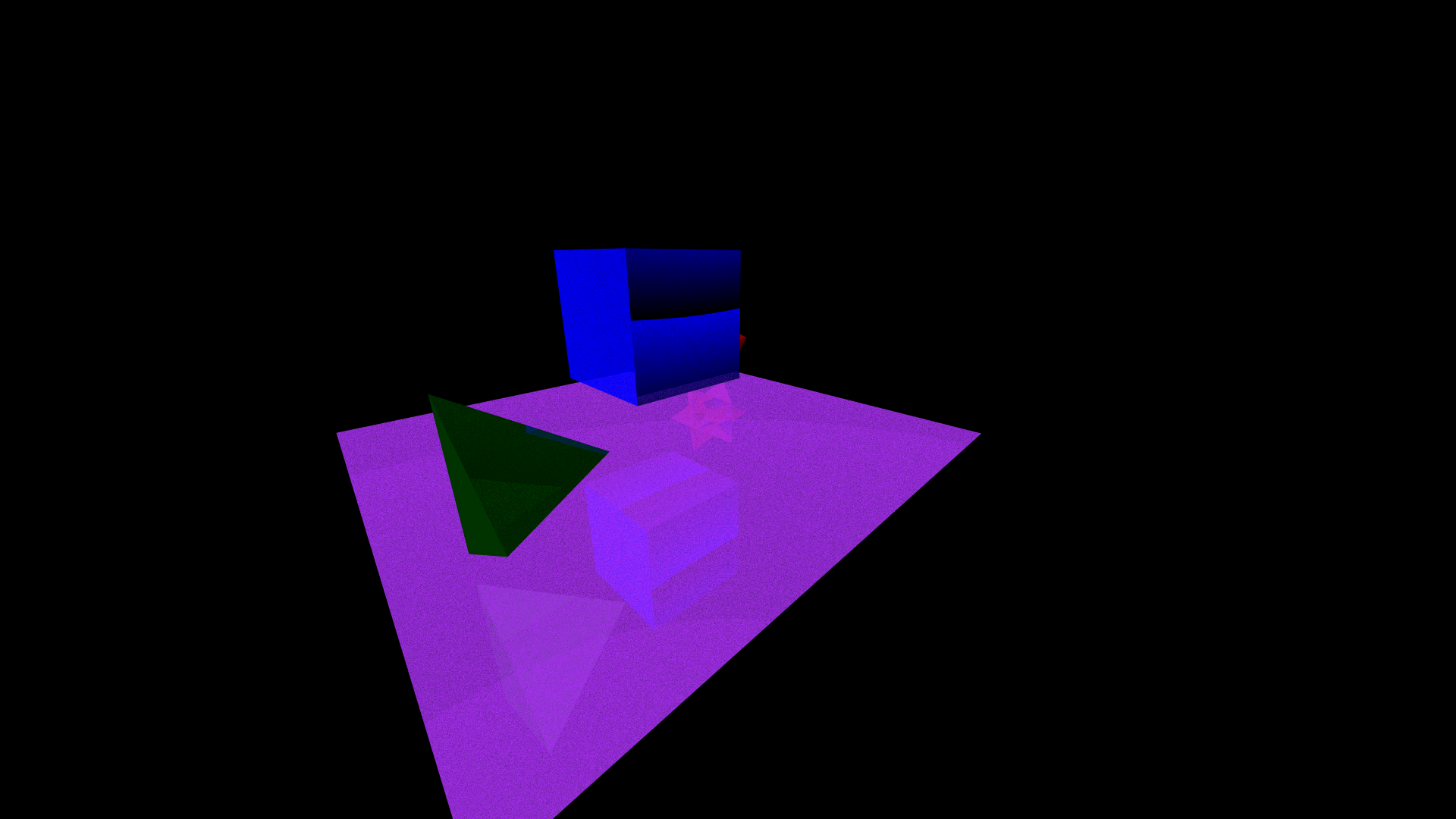
****

****

****

****

****

****

**Выводы**

Реализованный алгоритм трассировки лучей на C++ CUDA применяется в задачах фотореалистичного рендеринга, моделирования освещения, создания компьютерной графики и визуализации физических процессов. Типовые задачи включают расчет взаимодействия света с поверхностями, глобальное освещение, генерацию теней и отражений.

Программирование на CUDA потребовало оптимизации вычислений и управления памятью, что повысило скорость обработки сцен по сравнению с CPU-реализациями. Основные сложности включали управление потоками, балансировку нагрузки между блоками GPU и устранение артефактов (шум, недостаточное количество лучей).

Сравнение с традиционными методами показало значительное ускорение вычислений благодаря параллельной обработке лучей. Итоговые результаты продемонстрировали корректность алгоритма и возможность его масштабирования для более сложных сцен.

**Литература**

1. cppreference.com

2. http://www.ray-tracing.ru/