

ТЕХНОЛОГИЧНО УЧИЛИЩЕ ЕЛЕКТРОННИ СИСТЕМИ
към ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - СОФИЯ

ДИПЛОМНА РАБОТА

Тема: Програма за фотореалистична графика с трасиране на лъчи

Дипломант:

Борис Костадинов

Научен ръководител:

Александър Соклев

СОФИЯ

2021

Приложи подписаното задание тук.

Увод

В днешно време с експоненциалното нарастване на изчислителната мощност на видеокартите тежки алгоритми като raytrace навлизат в компютърната графика. Тези алгоритми за пръв път симулират правилно пътя на светлината в компютърното пространство като следват физичните закони на природата.

В голяма част от филмите и рекламите се използва този алгоритъм за симулиране на трудно постижими кадри и сцени. Някои болници симулират образа от рентгеновата снимка с изображение направено от алгоритъма, за да разберат кои тъкани е засегнала болестта. В игрите този raytrace не се използва заради сложността си и невъзможността на персоналните компютри да предоставят необходимия брой кадри в секунда. В бъдеще с нарастване на изчислителната мощност се очертава това да бъде основния начин за визуализация в игрите.

Тази дипломна работа има за цел да осъществи един базов raytrace алгоритъм, да прехвърли работата му на видеокарта. Алгоритъма ще има много функционалности. Камерата в алгоритъма ще има повечето функционалности на една истинска съвременна камера. Ще може да се добавят много лампи и различни обекти с текстури.

Първа глава

Методи и технологии за генериране на цифрово изображение в компютърната графика.

1.1. Основни принципи и техники в компютърната графика и визуализацията.

Рендъринг или рендериране е процесът на генериране на цифрово изображение (визуализация) от модел в компютърната графика. Под модел тук се има предвид описанието на всякакви обекти или явления, представени на строго определен математически език или във вид на масив от данни. Такова описание може да съдържа геометрични данни, гледната точка на наблюдателя, информация за осветлението, степента на наличие на някакво вещество, интензитет на физично поле и др. Цифровото изображение може да има вид на растрерна графика или на векторна графика. Рендъринг също се използва, за да се опише процеса на изчисляване на ефекти във видео файловете по време на монтаж или композиране и получаването на крайния видео продукт.

Рендеринг е един от основните предмети на тримерната компютърна графика и на практика е винаги свързан с останалите. То има за задача да пресъздаде математически модел (тримерен) върху плоска повърхност (двумерна) посредством различни математически алгоритми. Те се използват за постигането на перспектива в изображенията, за текстуриране, за оцветяване, за определяне на видимите обекти, за пресмятане на сенките, за моделиране на криви и огънати площи, за прозрачност и т.н.

Процесът започва с генерирането на моделите, след това се определя осветлението и посредством специални алгоритми за премахване на скрити повърхности се определят видимите за наблюдателя повърхности. Цветът на

всеки пиксел във видимата проекция е функция на отразената и излъчваната светлина от обектите.

За изобразяване на засенчванията се ползват математически алгоритми, които изчисляват междинните стойности на пикселите, изобразяващи повърхността на един многоъгълник или премахват границата между два съседни многоъгълника с еднаква осветеност.

В случая на тримерната графика, изобразяването може да се прави бавно, като предварително изобразяване с висока точност или в реално време. Предварителното изобразяване включва процес на интензивни изчисления и обикновено се използва при направата на филми, докато визуализирането в реално време се прави често за тримерни игри, които разчитат на употребата на графични карти с електронни тримерни ускорители.

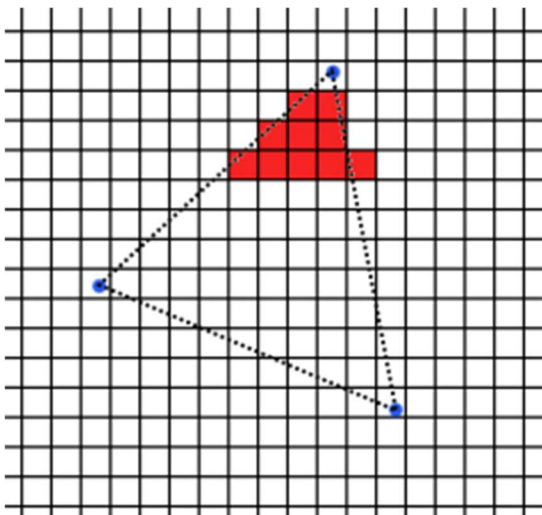
Визуализирането се прави, като към готово предварителното изображение (обикновено мрежова скица) се добавят растерни или процедурни текстури, светлини, картографиране на неравности и относително местоположение спрямо други обекти. Резултатът е завършено изображение, което клиентът или зрителят вижда.

1.2. Съществуващи решения и реализации.

1.2.1. Растеризация.

Това е най-разпространения вид алгоритъм, който се използва във видео картите като използва ресурса на тази хардуерна част чрез библиотеките OpenGL и Direct3D, които главно използват видео ускорителя, който много бързо изчислява матрични трансформации и да запълва екрана с изчислени цветове след направената трансформация. Този алгоритъм е изключително бърз, защото смята само промяната на цвета. Растеризацията използва триъгълници за представяне на триизмерните обекти в сцената. Триъгълника е

най-удобното тяло, защото при преобразуването му от двуизмерен в триизмерен и обратно той винаги запазва формата при проекцията си т.е. след перспективна трансформация той остава да бъде триъгълник. При растеризацията на триъгълник триизмерните координати на върховете на триъгълника се свеждат до двумерни екранни координати чрез подходяща перспективна трансформация, като се изрязват частите от триъгълника, които няма да се виждат (пример за това са тези които при рендериране излизат извън видимия екран). Тази трансформация е много бърза, защото трябва само да знаем цвета на триъгълника. Върховете се сортират по ординатата – горен, среден и долен. Обхождат се редовете между горния и средния. За всеки ред се намират най-левия и най-десният пиксел, който трябва да бъде оцветен. Редът се запълва с избрания цвят, след което се минава на следващия ред. Аналогично за редовете между средния и долния връх. Тези операции могат да се изпълняват изключително бързо на видеокарта.



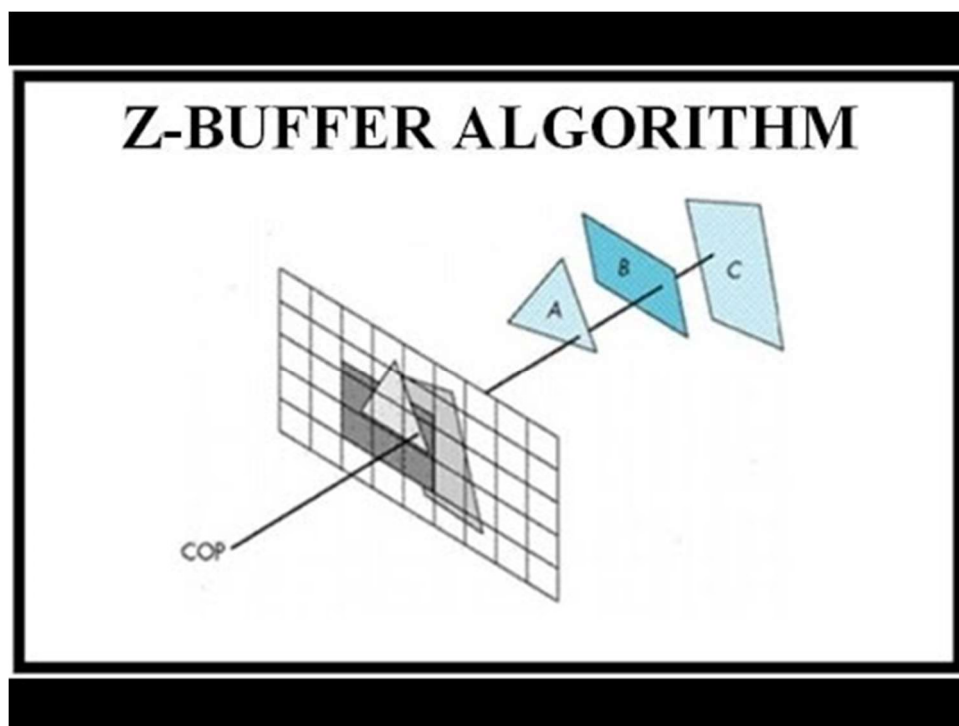
Фигура 1.1. Запълване на цветовете при растеризация.

При растеризацията обаче има проблем с това кой триъгълник се намира по-напред в сцената и трябва да бъде показан на крайната снимка. Първоначалния подход се е наричал „алгоритъм на художника“, в него първо са се изчертавали триъгълниците от най-далечния от камерата постепенно до

най-близкия като всеки следващ триъгълник е препокривал стария. Този алгоритъм е надежден, но бавен и тежък, защото всеки един триъгълник трябва да се изчертае. Другият използван алгоритъм в растеризацията е „Z-buffer“. При него първоначално за всеки пиксел се проверява най-близкия триъгълник и при намирането на такъв се взима цвета му. При този алгоритъм се решават няколко проблема: намалява се значително препокриването на триъгълници, лесно се проверява дали даден обект е видим, изчислителния процес се променя и в повечето случаи се печели от производителност.

Недостатъците на този алгоритъм са, че много ефекти няма как да бъдат симулирани. Например няма как да се симулират отражения и полупрозрачност, слънчеви зайчета и глобално осветление. Сенките са почти невъзможни. Няма как да се симулира снимка от физическа камера с лупа и изместване на фокуса (Depth Of Field effect). При много триъгълници алгоритъмът се забавя много.

Сложността се изчислява като $O(N)$, където N е броят на триъгълниците.



Фигура 1.2. Z-buffer algorithm

1.2.2. Трасиране на лъчи (Ray Tracing).

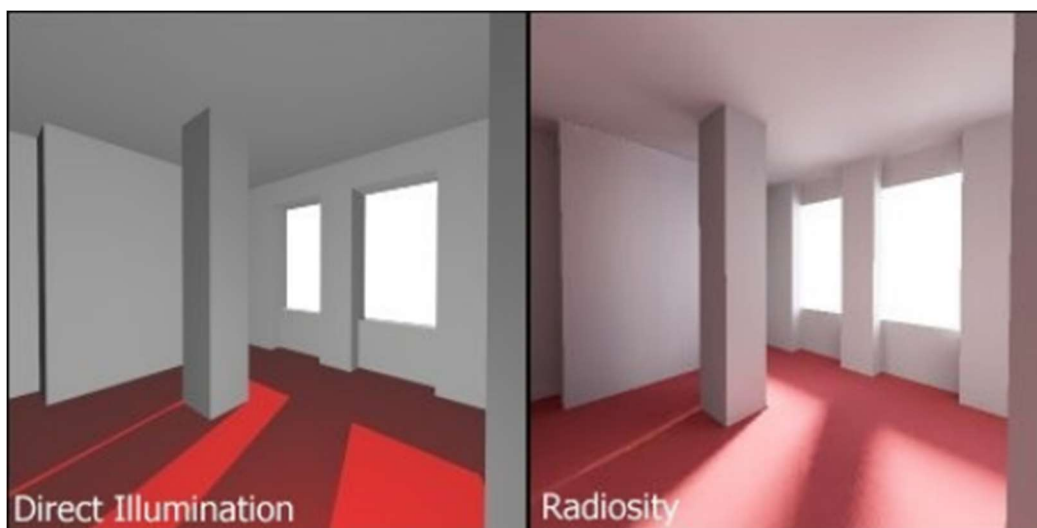
Този алгоритъм за рендериране смята пикселите последователно един след друг, като трасира лъчи и смята пресичането с различни триъгълници и геометрии. В природата фотоните тръгват от светлинните източници и стигат след пречупвания до човешкото око или камерата, а в рейтрейсинга лъчите започват от камерата и се разпространяват в сцената, като се проследява техният път. Това се нарича прав рейтрейсинг. Лъчите в този алгоритъм се държат като фотоните в природата – движат се по права линия, отразяват се от повърхности и се пречупват през различните материали. Резултатното изображение получаваме, като за всеки пиксел от него пуснем поне един лъч. За всеки лъч, проследяваме в каква повърхност се удрия първо и след което в зависимост от повърхността и осветеността, определяме цвета на пиксела. Ако усредним повече от един лъч през пиксел получаваме ефекта на преливане на цветовете на обектите в далечина и не толкова остри ръбове (Antialiasing). При сложни материали (например стъкло), рейтрейсингът ни позволява да разцепим лъча на две части – отразен и пречупен (както става и в природата). Това увеличава експоненциално броя лъчи, но обуславя висок реализъм. Лъчите стават много, както и изчислителната работа. Трябва да се трасира минимум по един лъч за всеки пиксел, а за проверяване на сенки броят, се увеличава многократно. Raytracing-ът е традиционно бавен алгоритъм и силата му е именно в офлайн рендерирането т.е не е подходящ за рендериране в реално време, но за в бъдеще с нарастване производителността на персоналните компютри това се очетвата да се промени. Този алгоритъм поддържа различни видове светлинни източници. Всеки обект може да се направи да излъчва светлина и така лесно можем да симулираме меки сенки.



Фигура 1.3. Светлина в Raytrace алгоритъма.

При рейтрейсинга, осветлението на дадена точка може да се изрази като сума приносите от видимите светлинни източници (директно осветление), но можем и да го изразим като сума от светлината, идваща от всички посоки, включително и други обекти, принципно не излъчващи светлина (глобално осветление), което допринася много за фото реалистичността на картината.

Глобалното осветление не е специфично само за рейтрейсинга, може да се приложи и при Z-buffer алгоритъма, но се изисква преизчисляване, за което пак обикновено се използва рейтрейсинг.



Фигура 1.4. Разликата между точкова светлина и глобално осветление.

Интересен факт е, че рейтрейсинга на теория може да направи „по-реалистична симулация на вселената“. За пример - една малка свещ излъчва

около 10^{17} фотона за секунда. Ако за всеки фотон трасираме по един лъч ще са нужни над 3 години за да се направи реалистична неразличима с истинска картинка (при обработка на RTX2080Ti).

Недостатъците на Raytracing-а са на първо място в скоростта, която засега е най-голямата пречка той да не е основния алгоритъм за рендериране. Алгоритмите са тежки, лесно е да се объркат и е трудно да се провери дали работят вярно. Сцените с много обекти изискват трасиране на много лъчи и отнемат огромно количество памет.

Сложност на raytracing алгоритъма. Приемаме N за брой обекти (елементарни примитиви) и M за брой пиксели. При пресичането на лъч със сцената, трябва да се обхождат всички обекти, за да се провери коя пресечна точка е най-близка. Сложността е $O(N \cdot M)$ при наивна имплементация. Но съществуват дървета за бързо пресичане, чрез които търсенето става логаритмично и сложността става $O(M \cdot \log(N))$.

1.2.3. Рендериране на сканирана линия (Scanline rendering).

При този алгоритъм основната идея е да се прекара една виртуална равнина, която да „отсича“ цялата сцена. Тази равнина съвпада с един ред пиксели от екрана, т.е. обектите, които пресича, са тези, които ще бъдат начертани на този ред от екрана. Поддържа се един списък с активни обекти (които пресичат сканиращата линия). При преминаване на следващ ред, този списък трябва да се обнови (т.е. някои от обектите отпадат, а се появяват някои нови). Списъкът се пази сортиран по абсциса (X координатите) на ръбовете на обектите. Рендера на единичен ред се състои от обхождане на списъка за определяне на видимите обхвати. Те се изчертават с подходящите цветове, подобно на растеризацията при Z-buffer.

Предимствата на този алгоритъм са, че не е необходимо да се пазят всички обекти в паметта, достатъчно е да се сортирани по Y – алгоритъмът ги взима един по един, като никога не се връща назад. Поради същата причина, алгоритъмът разглежда даден връх само веднъж. Няма припокриване (overdraw) – всеки пиксел се разглежда еднократно.

Недостатъците са че ако обектите са много, временните структури (списъка с активните обекти) стават големи и достигат размера на пълен Z-buffer. В такъв случай този алгоритъм губи преимуществото си пред Z-buffer. Алгоритъмът е труден за паралелизиране и не се поддържа от видео хардуера. Споделя повечето от недостатъците на Z-buffer. Той е един от първите алгоритми за видео обработка.

1.2.4. Сравнение.

Z-buffer е бърз, прост алгоритъм, с добра хардуерна поддръжка. Времето за рендериране при Z-buffer расте линейно спрямо броя триъгълници. Разделителната способност не оказва голямо влияние. При добра имплементация, времето за рендериране при Raytracing расте логаритмично спрямо броя триъгълници, и линейно спрямо разделителната способност. Тоест Raytracing-ът има капацитета да се справи с огромно количество триъгълници, и след даден момент, започва да се справя по-добре от Z-buffer. Raytrace алгоритъма се паралелизира по-добре от другите два алгоритъма (това обуславя ползването на рендер-ферми). Scanline алгоритъма е остарял и не предоставя възможностите на другите алгоритми, затова почти не се ползва. Поради появата на библиотеките с общо предназначение за видеокартите (като CUDA и OpenCL), бариерите между raytracing алгоритма и растеризация хардуер се разминават.

Втора глава

Функционални изисквания към алгоритъма. Избор на софтуерни средства. Проектиране структурата на алгоритъма.

2.1. Функционални изисквания към алгоритъма за фотореалистична графика.

- Поддръжка на зареждане на сцена от текстови файл
- Поддръжка на точкова (pinhole) камера
- Поддръжка на различни видове геометрия - реализацията на триъгълните мрежи ще е с библиотеката OptiX Prime
- Поддръжка на различни видове лампи
- Поддръжка на видове материали
- Поддръжка на сглаждане (anti-aliasing)

2.2. Избор на софтуерни средства.

2.2.1. Интегрална среда „Visual Studio 2019“.

Microsoft Visual Studio е мощна интегрирана среда за разработка на софтуерни приложения за Windows. Използва се за разработка на конзолни и графични потребителски интерфейс приложения. Visual Studio предоставя мощна интегрирана среда за писане на код, компилиране, изпълнение, дебъгване (както за високо така и за машинно ниво), тестване на приложения, дизайн на потребителски интерфейс (форми, диалози, уеб страници, визуални контроли и други), моделиране на данни, моделиране на класове, изпълнение на тестове, пакетиране на приложения и други функции. Могат да се добавят и плъгини, които повишават функционалността на почти всяко ниво.

2.2.2. „C“ програмен език.

C (си) е език за програмиране от средно ниво с общо предназначение. Поради ниското ниво на абстракция, програмистите имат повече контрол върху хардуера и програмите написани на него обикновено работят по-бързо от тези, написани на езици от високо ниво. Затова C е подходящ за създаване както на операционни системи, така и на приложения.

2.2.3. „C++“ програмен език.

C++ е неспециализиран език за програмиране от високо ниво. Той е обектно-ориентиран език със статични типове. Този език за програмиране е създаден като разширение на езика C – езикът е базиран на C, но в него са добавени редица допълнителни възможности и са направени няколко промени. Основната разлика между C и C++ е, че C++ съдържа вградена в езика поддръжка на обектно-ориентирано програмиране. В C++ са добавени класове, множествено наследяване, виртуални функции, шаблони (templates), обработка на изключения (exceptions) и вградени оператори за работа с динамична памет.

2.2.4. „CUDA“ библиотека.

CUDA (Compute Unified Device Architecture) е библиотека за паралелно изчисляване, създадена от Nvidia. Тя позволява на разработчиците на софтуер и софтуерните инженери да използват видеокартата за обработка на информация (изчисления с общо предназначение на графични процесори). Платформата CUDA е софтуерен слой, който дава директен достъп до виртуалния набор от инструкции на видеокартата и паралелни изчислителни елементи за изпълнение на изчислителните ядра.

2.2.5. „OptiX“ библиотека.

Nvidia OptiX е библиотека за проследяване на лъчи. Изчисленията се разтоварват към графичните процесори че библиотеката CUDA. CUDA се предлага само за графичните продукти на Nvidia. Nvidia OptiX е част от Nvidia

GameWorks. OptiX е библиотека от високо ниво, тоест тя е проектирана да капсулира целия алгоритъм с трасирането на лъчи.

2.2.6. Софтуер за генериране на проекти „CMake“.

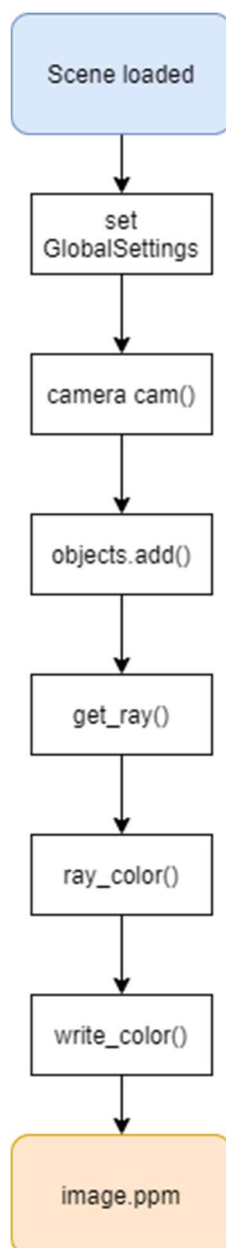
CMake е софтуер за автоматично генериране на проекти, независим от компилатора. CMake поддържа йерархии на директории и приложения, които зависят от множество библиотеки. CMake се използва заедно със среди за изграждане като Make, Qt Creator, Ninja, Android Studio, Xcode на Apple и Microsoft Visual Studio.

2.2.7. Визуализираща програма „IrfanView“.

IrfanView е програма за преглед на изображения, редактор, организатор и конвертор за Microsoft Windows. Той също така може да възпроизвежда видео и аудио файлове и има някои възможности за създаване на изображения и рисуване. IrfanView поддържа визуализация на PPM файлови формати и затова е избран за проекта.

2.3. Проектиране структурата на алгоритъма.

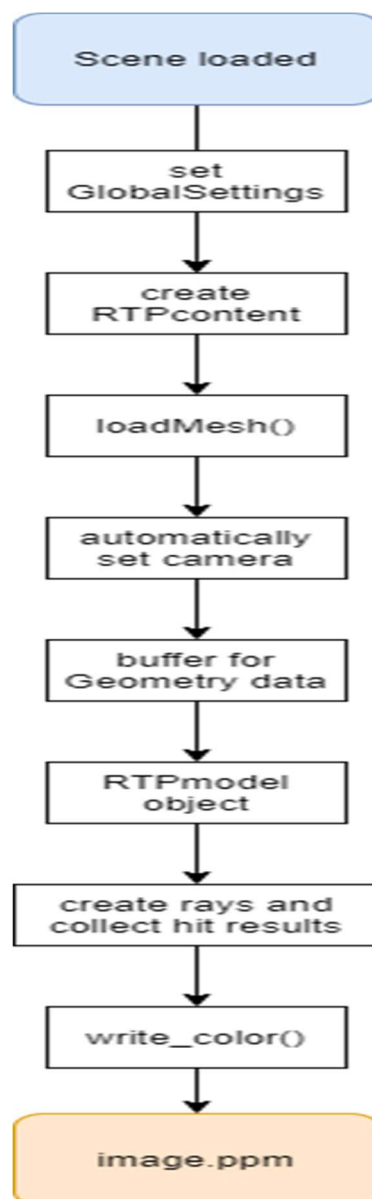
Алгоритъма чете текстови файл и сам създава средата на работа. Първата му работа е да провери дали е подаден триизмерен обект създаден от множество триъгълници. Ако не е подаден алгоритъма взема стойностите подадени във файла с информация за сцената и създава камера и определя размера на изображението. След това зарежда подадените в текстовия файл текстури и обекти. Създава лъчи с начало камерата и ги пресича с обекти в сцената като резултата от това пресичане се смята и визуализира под формата на пореден пиксел в крайното изображение. За един пиксел се изстрелват толкова лъчи, колкото потребителя е задал под променливата `samples_per_pixel`. На фигура 2.1 е показана блок схемата на тази част от алгоритъма.



Фигура 2.1. Блок схема на алгоритъма без зареждане на триизмерен обект.

Ако потребителя иска визуализация на триизмерен обект алгоритъма трябва да приеме променливи свързани с определянето на средата на работа от текстовия файл. След това трябва да създаде подходящо съдържание за зареждане на библиотеката OptiX Prime за пресичане на лъчи, изчислявани на видео картата. След това трябва да зареди този триизмерен обект и автоматично да сложи камерата на подходящо място за визуализация. Автоматичното създаване на камера е нужно поради причината, че всеки триизмерен обект е с различна форма и в почти всички случай камерата не вижда обекта – или е в

него, или той е прекалено малък и отдалечен и крайния резултат е незадоволителен. Следващата стъпка е създаване на буфер съдържащ геометричните данни от триизмерния файл. С всичките данни до тук библиотеката създава сцена в която се създават лъчи с начало камерата и се смята тяхното пресичане с триъгълник от триизмерния обект. Резултата се визуализира като пореден пиксел в крайното изображение. След смятането на всички лъчи се освобождава вече ненужната информация за триизмерния обект и създадената сцена.



Фигура 2.2. Блок схема на алгоритъма при зареждане на триизмерен обект.

Трета глава

Програмна реализация на алгоритъма.

3.1. Клас за вектори „vec3“.

Това е основния клас в проекта. Той съдържа три координати. Използва се както за цвят (във формата RGB), така и за местоположение, посока и отместване. За по-лесна работа при използване на цвят извикваме класа под името „color“, а за точка в пространството с „point3“.

```
class vec3 {
public:
    vec3() : e{ 0,0,0 } {}
    vec3(double e0, double e1, double e2) : e{ e0, e1, e2 } {}

    double x() const { return e[0]; }
    double y() const { return e[1]; }
    double z() const { return e[2]; }

    vec3 operator-() const { return vec3(-e[0], -e[1], -e[2]); }
    double operator[](int i) const { return e[i]; }
    double& operator[](int i) { return e[i]; }
public:
    double e[3];
}
```

Фигура 3.1. Header file vec3.h.

3.2. Лъчи.

Класа за лъчи - ray, се състои от начална точка и посока в която се разпространява.

```
#include "vec3.h"

class ray {
public:
    ray() {}
    ray(const point3& origin, const vec3& direction, double time =
0.0)
        : orig(origin), dir(direction), tm(time)
    {}

    point3 origin() const { return orig; }
    vec3 direction() const { return dir; }
    double time() const { return tm; }

    point3 at(double t) const {
        return orig + t * dir;
    }

public:
    point3 orig;
    vec3 dir;
    double tm;
};
```

Фигура 3.2. Клас за лъчи „ray”.

3.3. Абстрактния клас *hittable* съдържащ обектите в сцената.

Този клас съдържа булевата функция `hit()`, която взима лъч и връща отговор дали се е пресякъл с даден обект в сцената.

```
class material;

struct hit_record {
    point3 p;
    vec3 normal;
    shared_ptr<material> mat_ptr;
    double t;
    double u;
    double v;
    bool front_face;

    inline void set_face_normal(const ray& r, const vec3&
outward_normal) {
        front_face = dot(r.direction(), outward_normal) < 0;
        normal = front_face ? outward_normal : -outward_normal;
    }
};

class hittable {
public:
    virtual bool hit(const ray& r, double t_min, double t_max,
hit_record& rec) const = 0;
    virtual bool bounding_box(double time0, double time1, aabb&
output_box) const = 0;
};
```

Фигура 3.3. Абстрактния клас „hittable“.

3.4. Клас „hittable_list“.

Обектът (наричен hittable) може да се пресече с лъч, класът hittable_list пази лист от всички тези пресичания.

```
class hittable_list : public hittable {
public:
    hittable_list() {}
    hittable_list(shared_ptr<hittable> object) { add(object); }

    void clear() { objects.clear(); }
    void add(shared_ptr<hittable> object) {
objects.push_back(object); }

    virtual bool hit(
        const ray& r, double t_min, double t_max, hit_record& rec)
const override;

    virtual bool bounding_box(
        double time0, double time1, aabb& output_box) const
override;

public:
    std::vector<shared_ptr<hittable>> objects;
};
```

Фигура 3.4. Клас „hittable_list.h“.

3.5. Клас „camera“.

В този клас се създава точкова камера. Тя трябва да съдържа информация нужна за визуализация на сцената. Съдържа ширината и височината в пиксели на изображението, както и съотношението между тях. Задава се точката в която се намира и посоката в която гледа, разстояние за фокусиране, информацията за блендата ако има такава и обхвата на заснемане в градуси.

Дълбочина на рязкост или просто рязкост (още дълбочина на рязко изобразеното пространство, дълбочина на остротата, дълбочина на фокуса) (на

английски: Depth of Field - DOF), е понятие използвано във фотографията. С него обозначаваме пространството пред фотообектива между най-близкия и най-далечния обект, който е на фокус в кадър. Дълбочината на рязкост намалява при по-отворена бленда и се увеличава при по-затворена бленда. Обекти „на фокус“ от нула до безкрайност се фотографират с възможно най-затворена бленда и с обектив с късо фокусно разстояние.

Функцията `get_ray()` създава лъч с начало от камерата и посока зададена от параметрите в него.

```
class camera {
public:
    camera(
        point3 lookfrom,
        point3 lookat,
        vec3 vup,
        double vfov, // vertical field-of-view in degrees
        double aspect_ratio,
        double aperture,
        double focus_dist,
        double _time0 = 0,
        double _time1 = 0
    ) {
        auto theta = degrees_to_radians(vfov);
        auto h = tan(theta / 2);
        auto viewport_height = 2.0 * h;
        auto viewport_width = aspect_ratio * viewport_height;

        w = unit_vector(lookfrom - lookat);
        u = unit_vector(cross(vup, w));
        v = cross(w, u);

        origin = lookfrom;
        horizontal = focus_dist * viewport_width * u;
        vertical = focus_dist * viewport_height * v;
        lower_left_corner = origin - horizontal / 2 - vertical / 2
- focus_dist * w;

        lens_radius = aperture / 2;
        time0 = _time0;
    }
};
```

```

        time1 = _time1;
    }

    ray get_ray(double s, double t) const {
        vec3 rd = lens_radius * random_in_unit_disk();
        vec3 offset = u * rd.x() + v * rd.y();

        return ray(
            origin + offset,
            lower_left_corner + s * horizontal + t * vertical -
origin - offset,
            random_double(time0, time1));
    }

private:
    point3 origin;
    point3 lower_left_corner;
    vec3 horizontal;
    vec3 vertical;
    vec3 u, v, w;
    double lens_radius;
    double time0, time1; // shutter open/close times
};

```

Фигура 3.5. Клас „camera“.

3.6. Сглаждане „anti-aliasing“.

Сглаждането се получава като за всеки пиксел от изображението се пускат предварително зададен брой лъчи и се смята средно аритметичния резултат от получения цвят при присичането на всеки лъч с обект от сцената.

Функцията *write_color()* пише във файл цвета на всеки пиксел.

```

        color pixel_color(0, 0, 0);
    for (int s = 0; s < samples_per_pixel; ++s) {
        auto u = (i + random_double()) / (image_width - 1);
        auto v = (j + random_double()) / (image_height - 1);
        ray r = cam.get_ray(u, v);
        pixel_color += ray_color(r, background, objects,
max_depth);
    }
    write_color(file, pixel_color, samples_per_pixel);

```

Фигура 3.6. Сглаждане.

3.7. Клас „material”.

За да различни видове материали е нужен абстрактния клас `material`. Неговите цели са 2 - да произведе разпръскването на лъча (или да каже че материала го е абсорбирал) и ако лъча е разпространен да върне колко е отслабнала силата му след първия удар с повърхността на материала.

Функцията `scatter()` изчислява разпръскването на лъчите.

Функцията `color emitted()` изчислява разпръскването на светлина.

```

    struct hit_record;

    class material {
    public:
        virtual color emitted(double u, double v, const point3& p)
const {
            return color(0, 0, 0);
        }

        virtual bool scatter(
            const ray& r_in, const hit_record& rec, color&
attenuation, ray& scattered
        ) const = 0;
    };

```

Фигура 3.7. Абстрактния клас за материали.

3.7.1. Матов материал (Lambertian color).

Този материал се характеризира с това, че не пречупва светлината, но може да погълне част от интензитета ѝ.

```
class lambertian : public material {
public:
    lambertian(const color& a) :
albedo(make_shared<solid_color>(a)) {}
    lambertian(shared_ptr<texture> a) : albedo(a) {}

    virtual bool scatter(
        const ray& r_in, const hit_record& rec, color&
attenuation, ray& scattered
    ) const override {
        auto scatter_direction = rec.normal +
random_unit_vector();

        // Catch degenerate scatter direction
        if (scatter_direction.near_zero())
            scatter_direction = rec.normal;

        scattered = ray(rec.p, scatter_direction, r_in.time());
        attenuation = albedo->value(rec.u, rec.v, rec.p);
        return true;
    }

public:
    shared_ptr<texture> albedo;
};
```

Фигура 3.8. Клас за матова повърхност.

3.7.2. Клас „metal“.

Металът се характеризира със своя цвят, способност да отразява светлината и коефициент за това колко добре е шлифован. Колкото по-добре е шлифован един метал, толкова повече прилича на огледало и обратното, ако не е шлифован се доближава до свойствата на матовия материал.

```

class metal : public material {
public:
    metal(const color& a, double f) : albedo(a), fuzz(f < 1 ? f :
1) {}

    virtual bool scatter(
        const ray& r_in, const hit_record& rec, color&
attenuation, ray& scattered
    ) const override {
        vec3 reflected = reflect(unit_vector(r_in.direction()),
rec.normal);
        scattered = ray(rec.p, reflected + fuzz *
random_in_unit_sphere(), r_in.time());
        attenuation = albedo;
        return (dot(scattered.direction(), rec.normal) > 0);
    }

public:
    color albedo;
    double fuzz;
};

```

Фигура 3.9. Клас симулиращ метална повърхност.

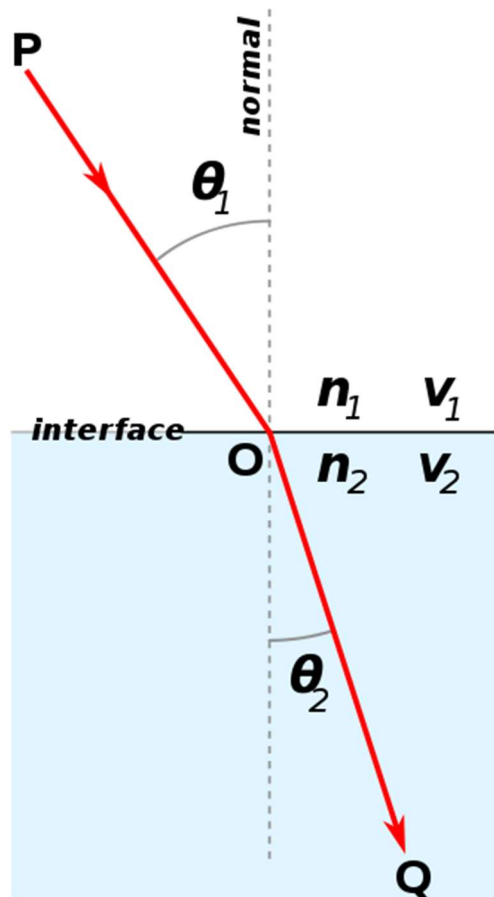
3.7.3. Клас симулиращ стъкло.

Стъклото пречупва светлината. Във физиката това явление е обяснено със закона на Снелиус. Законът на Снелиус описва пречупването на светлината на границата между две прозрачни среди.

Законът е открит в началото на XVII век от холандския математик Вилеброрд Снелиус Малко по-късно е публикуван от Рене Декарт.

Ъгълът на падане на светлината върху повърхността е свързан с ъгъла на пречупване чрез съотношението (3.1) където θ е ъгъла между нормалата и лъча.

$$\eta \cdot \sin\theta = \eta' \cdot \sin\theta' \quad (3.1)$$



Фигура 3.10 Закон на Снелиус

За да намерим посоката на пречупения лъч трябва да решим уравнението за :

От страната на пречупването имаме пречупен лъч R' и нормала n' . Получава се уравнението за R' :

$$R' = R'_{\perp} + R'_{\parallel} \quad (3.2)$$

Решаваме по отделно за двете събираеми :

$$\begin{aligned}\mathbf{R}'_{\perp} &= \frac{\eta}{\eta'}(\mathbf{R} + \cos \theta \mathbf{n}) \\ \mathbf{R}'_{\parallel} &= -\sqrt{1 - |\mathbf{R}'_{\perp}|^2} \mathbf{n}\end{aligned}\tag{3.3}$$

Заместваме нормалата и полученото уравнение е:

$$\mathbf{R}'_{\perp} = \eta/\eta'(\mathbf{R} + (-\mathbf{R} \cdot \mathbf{n})\mathbf{n})\tag{3.4}$$

Това е получената функция за отражение:

```
vec3 refract(const vec3& uv, const vec3& n, double
etai_over_etat) {
    auto cos_theta = fmin(dot(-uv, n), 1.0);
    vec3 r_out_perp = etai_over_etat * (uv + cos_theta * n);
    vec3 r_out_parallel = -sqrt(fabs(1.0 -
r_out_perp.length_squared())) * n;
    return r_out_perp + r_out_parallel;
}
```

Фигура 3.11. Функцията симулираща отражение.

3.7.4. Светлина.

За светлина ще използваме просто цвят, който надхвърля нормалните стойности. И чрез функцията *emitted()* ще изчисляваме коефициент на осветеност.

```
class diffuse_light : public material {
public:
    diffuse_light(shared_ptr<texture> a) : emit(a) {}
    diffuse_light(color c) : emit(make_shared<solid_color>(c)) {}

    virtual bool scatter(
        const ray& r_in, const hit_record& rec, color&
```

```

attenuation, ray& scattered
    ) const override {
        return false;
    }

    virtual color emitted(double u, double v, const point3& p)
const override {
    return emit->value(u, v, p);
}

public:
    shared_ptr<texture> emit;
};

```

Фигура 3.12. Класа симулиращ светлина.

3.8. Работа с библиотека „OptiX“.

Първоначално алгоритъма създава своя контекст. Тоест проверява и подготвя процесора и видеокартата (ако е съвместима) за работа.

```

RTPcontexttype contextType;
RTPbuffertype bufferType;
int runOnCpu = 0;
if (runOnCpu) {
    contextType = RTP_CONTEXT_TYPE_CPU;
    bufferType = RTP_BUFFER_TYPE_HOST;
}
else {
    contextType = RTP_CONTEXT_TYPE_CUDA;
    bufferType = RTP_BUFFER_TYPE_CUDA_LINEAR;
}

RTPcontext context;
if (runOnCpu) {
    CHK_PRIME(rtpContextCreate(contextType, &context));
}
else {

```

```

        CHK_PRIME(rtpContextCreate(contextType, &context));
        const unsigned deviceNumbers[] = { 0 };
        CHK_PRIME(rtpContextSetCudaDeviceNumbers(context, 1,
deviceNumbers));
    }

```

Фигура 3.13. Optix Content create

След създаване на контекст се зарежда триизмерния обект и камерата се позиционира автоматично спрямо размерите и местоположението на обекта. Информацията от този файл се форматира и съхранява в два буфера:

```

RTPbufferdesc indicesDesc;
CHK_PRIME(rtpBufferDescCreate(
    context,
    RTP_BUFFER_FORMAT_INDICES_INT3,
    bufferType,
    mesh.getVertexIndices(),
    &indicesDesc)
);
CHK_PRIME(rtpBufferDescSetRange(indicesDesc, 0,
mesh.num_triangles));

```

Фигура 3.14. Информация за индексите на триъгълниците.

```

RTPbufferdesc verticesDesc;
CHK_PRIME(rtpBufferDescCreate(
    context,
    RTP_BUFFER_FORMAT_VERTEX_FLOAT3,
    bufferType,
    mesh.getVertexData(),
    &verticesDesc)
);
CHK_PRIME(rtpBufferDescSetRange(verticesDesc, 0,
mesh.num_vertices));

```

Фигура 3.15. Информация за върховете на триъгълниците.

От тези буфери се създава и зарежда обект в сцената:

```
RTPmodel model;  
CHK_PRIME(rtpModelCreate(context, &model));  
CHK_PRIME(rtpModelSetTriangles(model, indicesDesc,  
verticesDesc));  
CHK_PRIME(rtpModelUpdate(model, RTP_MODEL_HINT_ASYNC));
```

Фигура 3.16. Създаване на обект в сцената.

Създава се масив (vector) от всички лъчи, който бива подаден на библиотеката, която трасира и смята къде тези лъчи се удрят с триъгълник от триизмерния обект - *Buffer<Ray> raysBuffer*. Библиотеката връща в подобен масив резултата от пресичането на лъчите - *Buffer<Hit> hitsBuffer*.

```
RTPquery query;  
CHK_PRIME(rtpQueryCreate(model, RTP_QUERY_TYPE_CLOSEST,  
&query));  
CHK_PRIME(rtpQuerySetRays(query, raysDesc));  
CHK_PRIME(rtpQuerySetHits(query, hitsDesc));  
CHK_PRIME(rtpQueryExecute(query, 0 /* hints */));
```

Фигура 3.17. Изпълнение на заявката.

След като OptiX е приключил работа снимката се визуализира, а информацията за триизмерния обект и контекста се освобождава.

Четвърта глава

Ръководство на потребителя.

4.1. Инсталиране на проекта.

4.1.1. Инсталиране на „Visual Studio 2019“.

Microsoft Visual Studio е мощна интегрирана среда за разработка на софтуерни приложения за Windows и можете да я изтеглите от официалния сайт на Microsoft или от този линк - <https://visualstudio.microsoft.com/downloads/>.

4.1.2. Инсталиране на „IrfanView“.

IrfanView е безплатна програма за преглед на изображения. Можете да използвате програма по ваш избор за визуализация на PPM файлов формат. За да изтеглите програмата отидете на официалния сайт на IrfanView - <https://www.irfanview.com/>.

4.1.3. Инсталиране на „CMake“.

CMake е безплатен софтуер за автоматично генериране на проекти и с него ще бъде създаден проекта. Нужна е версия 3.20.0 или по-нова. Може да бъде изтеглен от официалния сайт - <https://cmake.org/download/>.

4.1.4. Инсталиране на библиотека „OptiX 6.5.0“.

Тази библиотека е свободна за ползване от разработчици на софтуер. За достъпа до нея е нужна само регистрация в сайта на Nvidia Developer. Версия 6.5.0 може да бъде изтеглена от този линк - <https://developer.nvidia.com/designworks/optix/download>.

4.1.5. Инсталиране на „Cuda“.

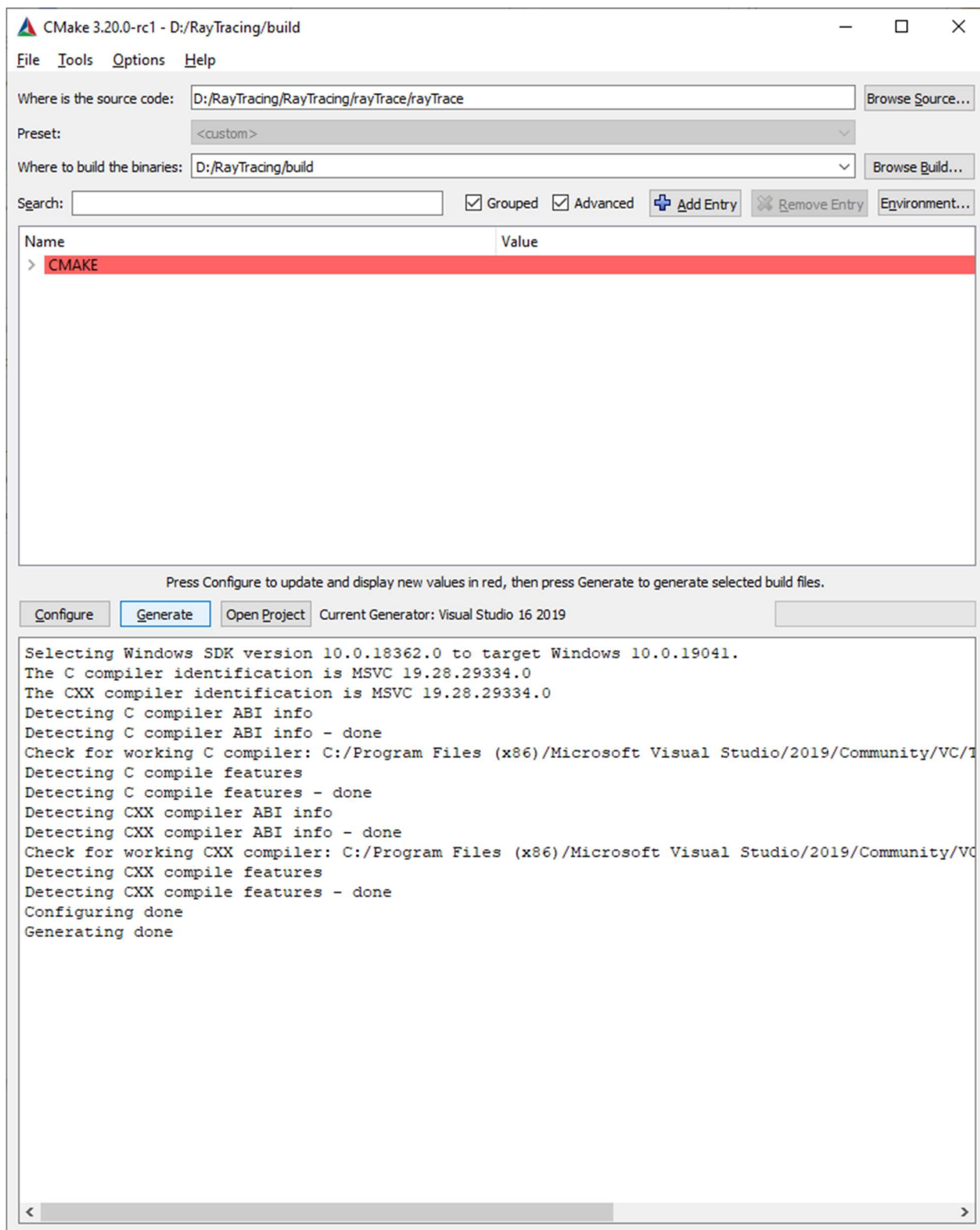
Cuda е библиотека за паралелно изчисляване, създадена от Nvidia. Тя позволява на разработчиците на софтуер и софтуерните инженери да използват видеокартата за обработка на информация. Нужна е версия 11.2.0, която може да бъде изтеглена от сайта на Nvidia Developer - <https://developer.nvidia.com/cuda-downloads>.

4.1.6. Клонирание на проекта от „GitHub“.

За да се клонира проекта отидете на <https://github.com/boriskostadinov/RayTracing> и изтеглете папката rayTrace. Кода на проекта се намира на rayTrace/rayTrace.

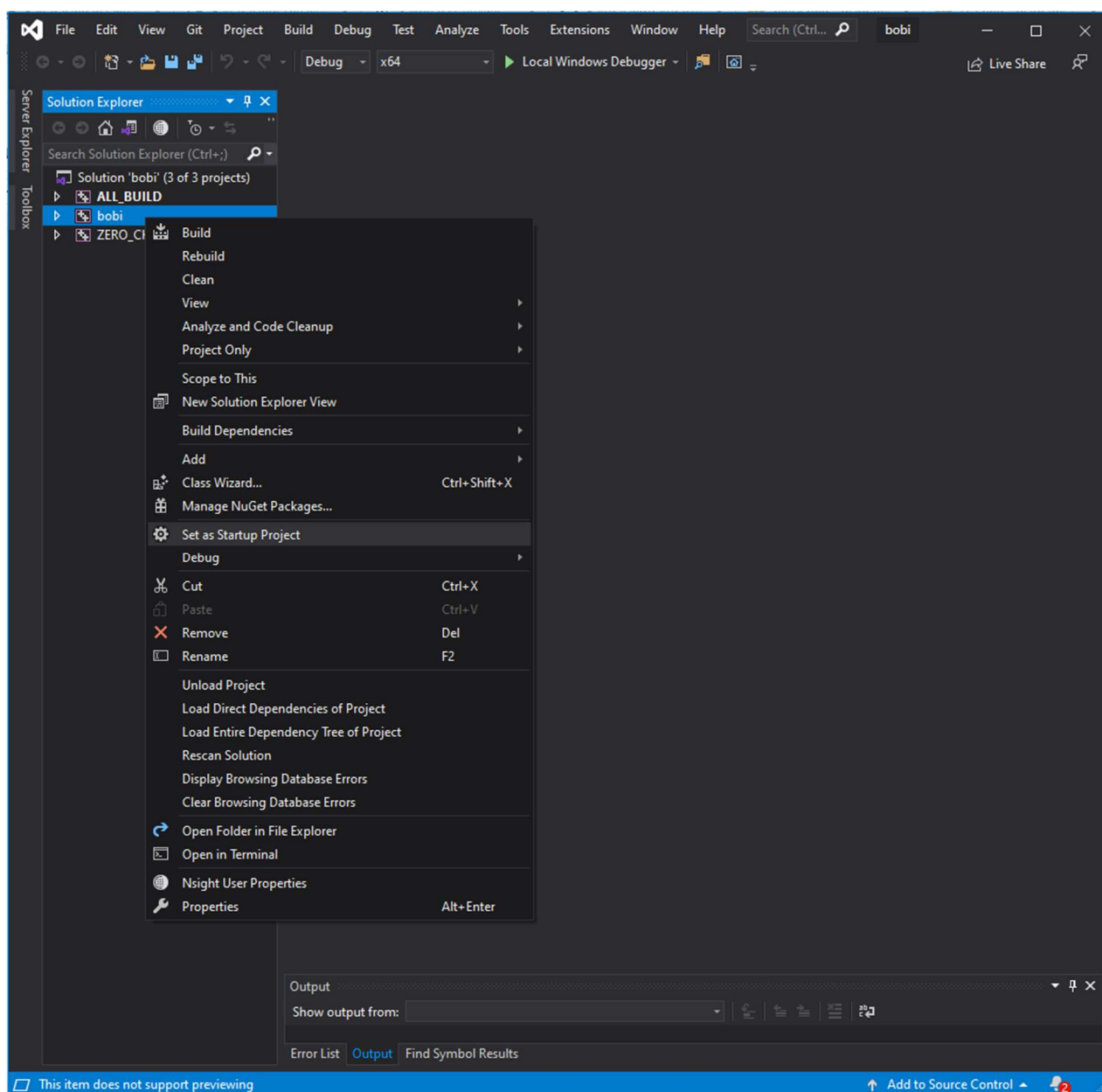
4.1.7. Създаване на проекта.

За да създадете проекта отворете CMake и в горното поле наречено „Where is the source code:“ поставете адреса на папката с кода на проекта. Уверете се, че там е файла CMakeLists.txt. Пример за това е D:\RayTracing\rayTrace\rayTrace. В третото поле наречено „Where to build the binaries:“ изберете къде искате да бъде създаден проекта. Пример за това е D:/RayTracing /build. От бутона в долния ляв ъгъл наречен „Configure“ конфигурирайте 2 пъти и след успешна конфигурация от бутона „Generate“ генерирайте проекта.



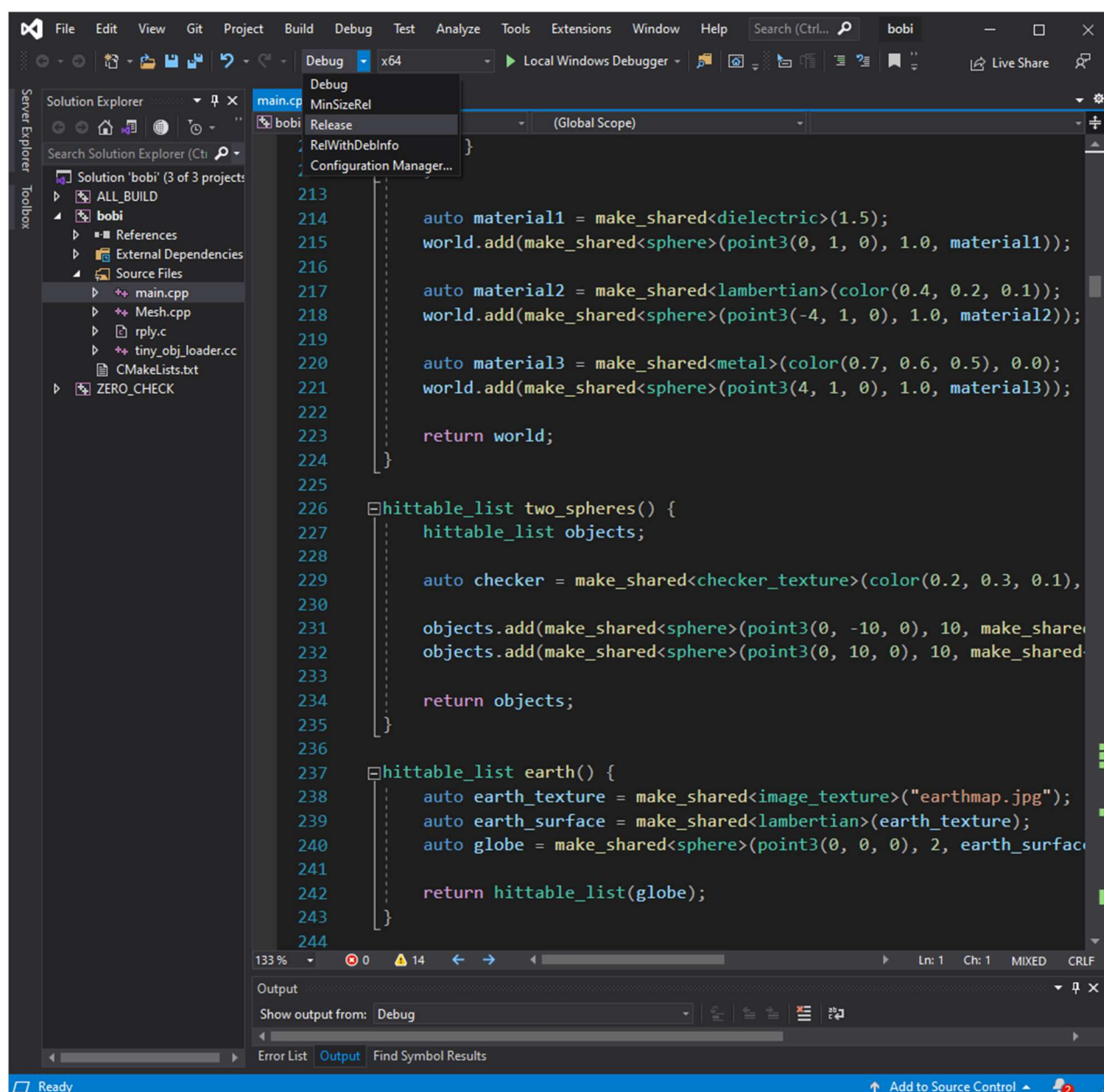
Фигура 4.1. Създаване на проекта в „CMake“.

Отворете папката на създадения проект и чрез Visual Studio отворете файла `bobi.sln`. От лентата с отворени проекти селектирайте проекта `bobi` и с десен бутон изберете „Set as Startup project“.



Фигура 4.2. Set as Startup project.

След това с клавиша F5 можете да компилирате и изпълните компилационния файл. За по-бързо изпълнение изберете от падащото меню опцията „Release“, така проекта се компилира с всички възможни компилации и изчисленията стават много по-бързо, следователно и изпълнимия файл е много по-малък. След успешна компилация и успешно завършване на изпълнимия файл изображението е готово и се намира в папката при изпълнимия файл.

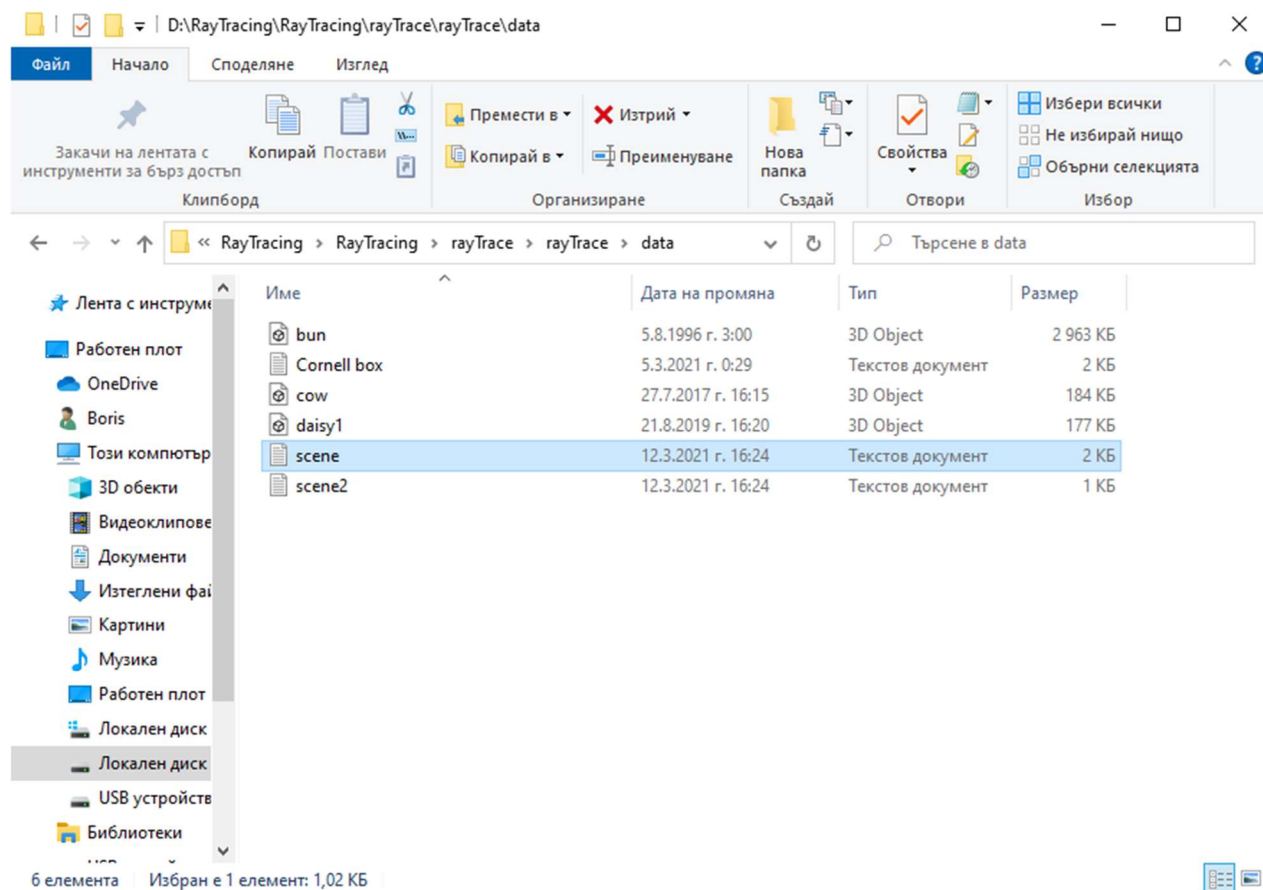


Фигура 4.3. Настройки на Visual Studio 2019.

4.2. Създаване на сцена.

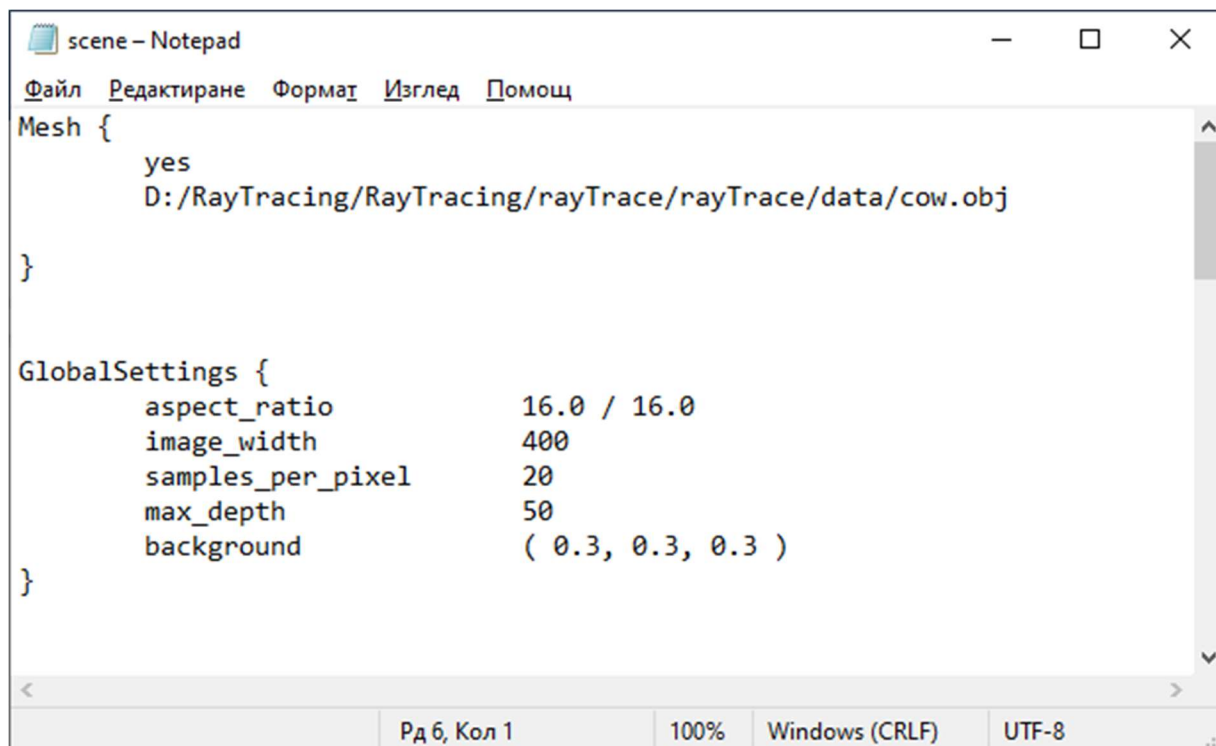
4.2.1. Сцена от текстови файл.

Алгоритъма взима за вход текстовия файл scene.txt, който се намира в проекта на този адрес \rayTrace\rayTrace\data.



Фигура 4.4. местоположение на текстовия файл за сцена.

В началото трябва да се избере дали зареждаме 3D обект и ако да да се даде пълния му път. Ако сме избрали такъв трябва да попълним само частта с глобалните настройки, всичко останало се изчислява автоматично, защото всеки 3D обект има различни размери и алгоритъма изчислява сам разстоянието до обекта и далечината на фокусната точка.

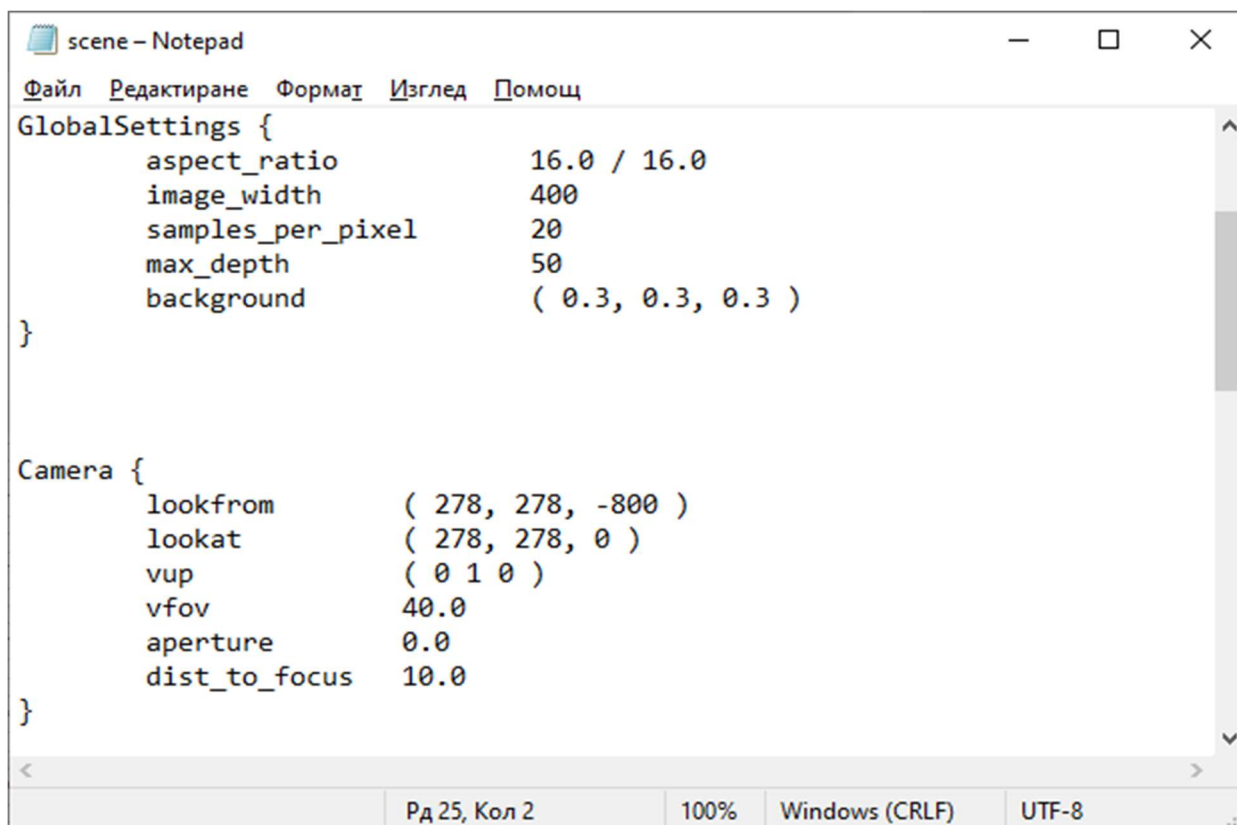


```
scene - Notepad
Файл Редактиране Формат Изглед Помощ
Mesh {
    yes
    D:/RayTracing/RayTracing/rayTrace/rayTrace/data/cow.obj
}

GlobalSettings {
    aspect_ratio      16.0 / 16.0
    image_width       400
    samples_per_pixel 20
    max_depth         50
    background        ( 0.3, 0.3, 0.3 )
}
```

Фигура 4.5. Примерни данни за 3D обект и глобални настройки.

Ако няма 3D обект следва да бъде зададена информация както за глобалните настройки така и за камерата. Пример за това е:



```
scene - Notepad
Файл Редактиране Формат Изглед Помощ
GlobalSettings {
    aspect_ratio      16.0 / 16.0
    image_width       400
    samples_per_pixel 20
    max_depth         50
    background        ( 0.3, 0.3, 0.3 )
}

Camera {
    lookfrom          ( 278, 278, -800 )
    lookat             ( 278, 278, 0 )
    vup                ( 0 1 0 )
    vfov               40.0
    aperture           0.0
    dist_to_focus      10.0
}
```

Фигура 4.6. Примерни данни за глобални настройки и камера.

След като средата е настроена трябва да се създадат обекти. Създаването на обекти става като се избере тип материал, от който да бъде обекта и неговите характеристики. След това се избира вида на обекта с неговите характеристики.

За оцветен обект се извиква `lambertian` и в скоби се поставят RGB стойностите на цвета (<https://www.tug.org/pracjourn/2007-4/walden/color.pdf> - примерни цветове) пример за това:

```
lambertian ( 0.65, 0.05, 0.05 ).
```

За метален обект е нужна ключовата дума `metal`, след това се подава цвят и се избира коефициент на шлифованост. Пример:

```
metal ( 0.8, 0.8, 0.8 ) 0.3.
```

За стъкло е нужна ключовата дума `dielectric` и коефициент на пречупване. Пример за това: `dielectric 1.5`.

За обект излъчващ светлина ключовата дума е `diffuse_light` със стойност на излъчване в тип подобен на RGB цвят. Пример за това:

```
diffuse_light ( 8 8 8 ).
```

След като сме избрали вида материал трябва да изберем и вида обект, което става по следния начин:

- За сфера трябва да подадем нейния център и радиус като

```
sphere ( 5 5 5 ) 1
```

- За куб трябва да подадем негови 2 точки по диагонал. Пример:

```
box ( 130, 0, 65 ) ( 295, 165, 230 )
```

- За стена трябва да се съобразим с нейната посока и съответно да изберем по коя основна посока е подравнена чрез една от ключовите думи: `yz_rect`, `xz_rect` и `xy_rect`, и 5 точки нужни за определянето ѝ в пространството.

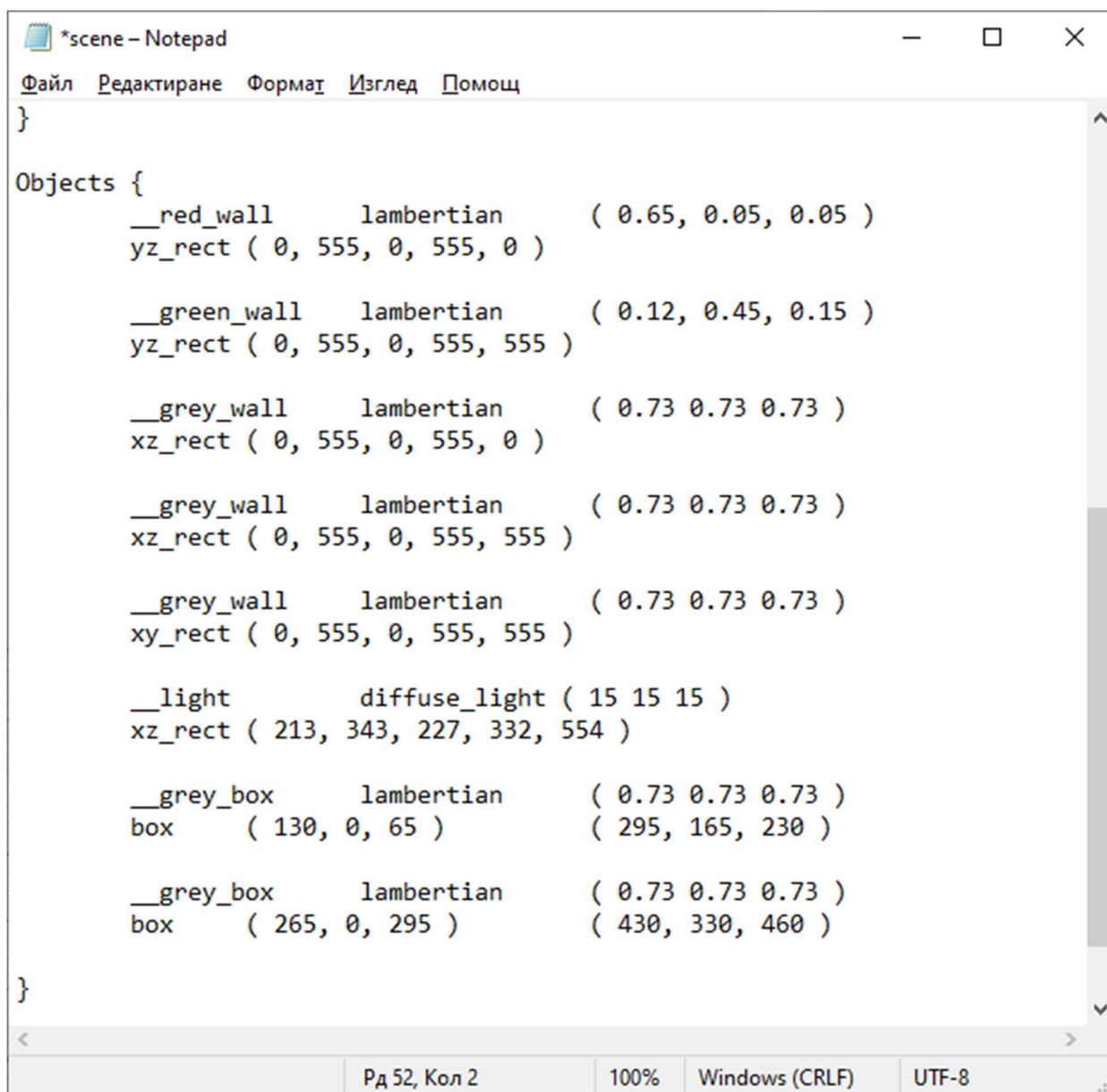
Примери за това са:

`yz_rect (0, 555, 0, 555, 0)`

`xz_rect (0, 555, 0, 555, 0)`

`xy_rect (0, 555, 0, 555, 555)`

Пример за добавяне на обекти е:



```
}  
  
Objects {  
    __red_wall    lambertian    ( 0.65, 0.05, 0.05 )  
    yz_rect ( 0, 555, 0, 555, 0 )  
  
    __green_wall  lambertian    ( 0.12, 0.45, 0.15 )  
    yz_rect ( 0, 555, 0, 555, 555 )  
  
    __grey_wall   lambertian    ( 0.73 0.73 0.73 )  
    xz_rect ( 0, 555, 0, 555, 0 )  
  
    __grey_wall   lambertian    ( 0.73 0.73 0.73 )  
    xz_rect ( 0, 555, 0, 555, 555 )  
  
    __grey_wall   lambertian    ( 0.73 0.73 0.73 )  
    xy_rect ( 0, 555, 0, 555, 555 )  
  
    __light       diffuse_light ( 15 15 15 )  
    xz_rect ( 213, 343, 227, 332, 554 )  
  
    __grey_box    lambertian    ( 0.73 0.73 0.73 )  
    box          ( 130, 0, 65 )  ( 295, 165, 230 )  
  
    __grey_box    lambertian    ( 0.73 0.73 0.73 )  
    box          ( 265, 0, 295 ) ( 430, 330, 460 )  
  
}
```

Фигура 4.7. Примерни данни за добавяне на обекти в сцената.

4.2.2. Добавяне на обекти в кода.

Обекти могат да бъдат добавени и в `main.cpp`. За целта ни трябва променлива от типа *hittable_list objects*. Нужна е и променлива за вида материал. Пример за това е:

```
auto per-text = make_shared<image_texture>("earthmap.jpg");
objects.add(make_shared<sphere>(point3(0, 2, 0), 2,
make_shared<lambertian>(per-text)));
auto checker = make_shared<checker_texture>(color(0.2, 0.3, 0.1),
color(0.9, 0.9, 0.9));
objects.add(make_shared<sphere>(point3(0, -10, 0), 10,
make_shared<lambertian>(checker)));
```

Фигура 4.8. Пример за добавяне на обекти.

Заклучение

Функционалните изисквания на дипломната работа са спазени и изпълнени. Алгоритъмът е готов за използване. Въпреки това разработката на Raytrace алгоритъм е дълъг и сложен процес. За да се постигне максимална реалистичност и бързина с всеки изминал ден в световен мащаб се измислят нови методи и функционалности за ускорение на алгоритъма.

Бъдещото развитие на проекта се състои в добавянето на още функционалности, като например:

- Паралелно изчисляване на видео карта и процесор;
- Зареждане на множество 3D обекта;
- Добавяне на нови материали и текстури;
- Monte Carlo алгоритъм за случайност (randomization);
- Глобално осветяване поддържано от видео картата.

Планирани са също подобрения на вече имплементираните функционалности, добавяне на визуализация в реално време чрез библиотеката SDL2 и други.

Използвана литература

1. Компютърно графично изобразяване -
[https://en.wikipedia.org/wiki/Rendering_\(computer_graphics\)](https://en.wikipedia.org/wiki/Rendering_(computer_graphics))
2. 3D компютърно графично изобразяване –
https://en.wikipedia.org/wiki/3D_rendering
3. Алгоритъм за трасиране на лъчи -
[https://en.wikipedia.org/wiki/Ray_tracing_\(graphics\)](https://en.wikipedia.org/wiki/Ray_tracing_(graphics))
4. Какво представлява алгоритъма за трасиране на лъчи -
<https://www.embedded.com/what-is-ray-tracing-and-how-is-it-enabling-real-time-3d-graphics/>
5. Курсът „3D графика и трасиране на лъчи“ към СУ-ФМИ -
<http://raytracing-bg.net/>
6. Език за програмиране „C“ -
[https://en.wikipedia.org/wiki/C_\(programming_language\)](https://en.wikipedia.org/wiki/C_(programming_language))
7. Език за програмиране „C++“ - <https://en.wikipedia.org/wiki/C%2B%2B>
8. Visual Studio 2019 - <https://docs.microsoft.com/en-us/visualstudio/get-started/visual-basic/tutorial-console?view=vs-2019>
9. Снимков формат ppm - <http://netpbm.sourceforge.net/doc/ppm.html>
10. Ламбертово отражение -
<https://www.sciencedirect.com/topics/computer-science/lambertian-model>
11. Закон на Снелиус - https://en.wikipedia.org/wiki/Snell%27s_law
12. Размазване на фокуса -
https://en.wikipedia.org/wiki/Defocus_aberration
13. Размазване в движение - https://en.wikipedia.org/wiki/Motion_blur

14. Картографиране на текстурата на изображението - https://en.wikipedia.org/wiki/Texture_mapping
15. Видеокарта - https://en.wikipedia.org/wiki/Graphics_processing_unit
16. Документация за програмиране на GPU - <https://docs.nvidia.com/cuda/>
17. Библиотеката за изобразяване на 3D обекти в сцена чрез GPU - https://raytracing-docs.nvidia.com/optix6/api_6_5/html/index.html
18. Подбиблиотеката OptiX Prime - https://raytracing-docs.nvidia.com/optix6/api_6_5/html/group__prime_reference.html
19. Първи стъпки с OptiX Prime - <https://www.programmingsought.com/article/58351467338/>
20. 3D обекти - <http://graphics.stanford.edu/data/3Dscanrep/>
21. Система за строене на проект CMake - <https://cmake.org/>

Съдържание

Увод	4
Първа глава	5
1.1. Основни принципи и техники в компютърната графика и визуализацията.	5
1.2. Съществуващи решения и реализации.	6
1.2.1. Растеризация.	6
1.2.2. Трасиране на лъчи (Ray Tracing).	9
1.2.3. Рендериране на сканирана линия (Scanline rendering).	11
1.2.4. Сравнение.	12
Втора глава	13
2.1. Функционални изисквания към алгоритъма за фотореалистична графика	13
2.2 Избор на софтуерни средства	13
2.2.1. Интегрална среда „Visual Studio 2019“.	13
2.2.2. „C“ програмен език.	14
2.2.3. „C++“ програмен език.	14
2.2.4. „CUDA“ библиотека.	14
2.2.5. „OptiX“ библиотека.	14
2.2.6. Софтуер за генериране на проекти „CMake“.	15
2.2.7. Визуализираща програма „IrfanView“.	15
2.3 Проектиране структурата на алгоритъма.	15
Трета глава	18
3.1. Клас за вектори „vec3“.	18
3.2 Лъчи.	19
3.3 Абстрактния клас hittable съдържащ обектите в сцената.	20
3.4. Клас „hittable_list“.	21
3.5 Клас „camera“.	21
	45

3.6. Сглаждане „anti-aliasing“.	23
3.7. Клас „material“.	24
3.7.1 Матов материал (Lambertian color)	25
3.7.2. Клас „metal“.	25
3.7.3. Клас симулиращ стъкло.	26
3.7.4. Светлина.	28
3.8. Работа с библиотека „OptiX“.	29
Четвърта глава	32
4.1 Инсталиране на проекта.	32
4.1.1. Инсталиране на „Visual Studio 2019“.	32
4.1.2. Инсталиране на „IrfanView“.	32
4.1.3 Инсталиране на „CMake“.	32
4.1.4 Инсталиране на библиотеката „OptiX 6.5.0“.	32
4.1.5 Инсталиране на „Cuda“.	33
4.1.6 Клонирание на проекта от „GitHub“.	33
4.1.7 Създаване на проекта.	33
4.2 Създаване на сцена.	37
4.2.1 Сцена от текстови файл.	37
4.2.2 Добавяне на обекти в кода.	41
Заключение	42
Използвана литература	43
Съдържание	45