Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Department of theoretical computer science

Bachelor's thesis

Parallelization of the raytracing method II

Michal Ďurovec

Supervisor: Ing., Mgr. Zdeněk Konfršt, PhD.

12th May 2015

Acknowledgements

I would like to express my thanks to my supervisor Dr. Zdeněk Konfršt for the advice and guidance he's given me.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 12th May 2015 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Ďurovec, Michal. *Parallelization of the raytracing method II.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

Abstrakt

Cílem této práce je nastudovat a implementovat metody počítačové grafiky o sledování paprsku. Tato práce zkoumá paralelizační techniky na GPU s využitím programovacího modelu NVIDIA CUDA a jazyku C++. Výsledkem této práce je jak funkční implementace raytraceru bežícího paralelně na GPU, tak i analýza jeho výkonnosti.

Klíčová slova Sledování paprsku, NVIDIA CUDA, renderer, paralelizace, C++, Počítačová grafika.

Abstract

The objective of this thesis is to study and implement ray tracing methods of computer graphics. It studies GPU-based parallelization techniques utilizing NVIDIA CUDA programming model and C++. The result of this thesis is a working implementation of a raytracer running in parallel on the GPU, as well as an analysis of its performance.

Keywords Ray tracing, NVIDIA CUDA, renderer, parallelization, C++, Computer graphics.

Contents

List of Figures

List of Tables

Introduction

Historically, there have been two main methods used for image rendering: Rasterization and Ray tracing. Rasterization goes from primitive to primitive (for example triangles, curves) and computes their position on screen by applying series of transformations to their coordinates. Ray tracing, on the other hand, tries a different approach and goes from pixel to pixel, tracing a ray coming through it from the eye and calculating its collisions with primitives in the scene. By tracing subsequent reflection and shadow rays it allows for a much more photorealistic result than rasterization. The downside is that it requires much more processing power, because the number of pixels is high and each pixel oftentimes involves calculating collisions for several recursively traced rays. Luckily, graphics hardware is built with parallelism in mind, where a single Graphics processing unit (GPU from now on) consists of hundreds, sometimes even thousands of cores capable of collectively running thousands of computing threads. This is perfect for ray tracing, as each pixel is independent from every other pixel, thus giving us an opportunity to run it in parallel very efficiently.

Overview of the thesis

The first chapter deals with the theory behind ray tracing by explaining the fundamental methods behind it and exploring some of the optimization possibilities. In Chapter 2 we look at GPU parallelization methods and how to structure our code to work with GPU using NVIDIA CUDA. Chapter 3 then delves into details and presents solutions to problems we encountered in our implementation. Lastly, we analyze and compare the results we got in Chapter 4.

Chapter 1

Ray Tracing

Ray tracing is a rendering technique working in pixel-order, where each image is rendered by tracing one (or more) rays through each pixel of the screen into the scene and computing the color of the pixel by figuring out which objects does the ray collide with and bringing their material properties, coordinates and data from other recursively traced rays into calculation.

1.1 Ray casting

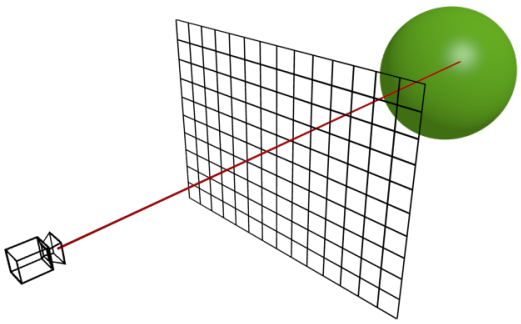

Figure 1.1: Simple scheme of ray casting. One ray is traced through every pixel of the image.

Ray casting could be considered the basis of ray tracing and was first developed by Arthur Appel, who used it for visible surface determination [7]. The raycaster is given a viewpoint as well as a view plane divided into grid, where each element represents a pixel of the final rendered image. After that, it generates a batch of primary rays, one for every pixel of the image. Each ray originates from the viewpoint and intersects the grid in the corresponding

pixel. It is then traced through the scene until the closest ray-object collision is found, as shown in Figure 1.1. The resulting color of the pixel was determined without any subsequent tracing of rays.

1.2 Shading

So far we have only talked about detecting whether ray hit an object and did not really cover how the final color of the pixel is calculated. If we were to just set the color of the pixel to a predetermined color of the object, the image would just be a series of flat shapes, which might not even resemble a 3D scene. The process of determining the final color of the pixel is called shading and there are many different algorithms covering this topic.

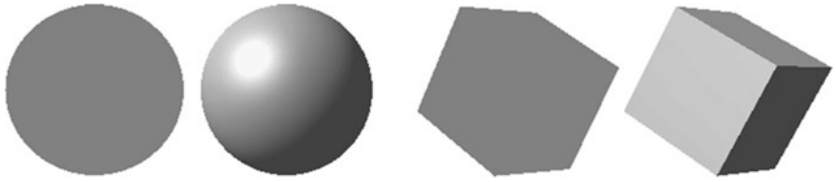

Figure 1.2: Example of shaded and unshaded objects. Adapted from [1].

The first sphere and cube are without shading, while the shaded sphere uses Phong shading and the cube is flat shaded.

1.2.1 Local and Global illumination

Local and global illumination represent two types of illumination models based on which we compute the shading of an object. The difference is that the local illumination adds direct contributions of every light in the scene towards the color of the object, while the global illumination also takes the interactions of the light with the other objects into account, thus creating effects like shadows and reflections.

1.2.2 Phong reflection model

One of the most common illumination models used in computer graphics is Phong reflection model, developed by Bui Tuong Phong in 1973 [8]. It is a form of local illumination, thus only taking into account properties of current object and all lights in the scene. The specific model we describe and use in our thesis is called Bling-Phong shading, which is a modification of the Phong shading developed by Jim Blinn. It is very similar with the only difference being in the way it calculates the specular highlights. That approach is based on adding three main light components to create realistically shaded results. First is an ambient reflection, which represents all the light coming from the

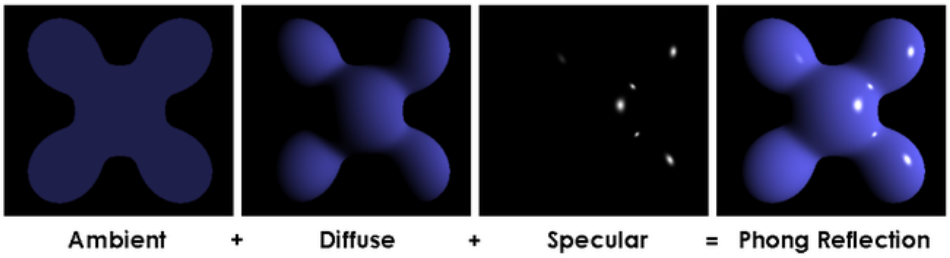

Figure 1.3: An example showing how the reflection components are combined in Phong illumination. Adapted from [2].

environment. It is just a single uniform color and could be understood as the base color of an object, without which all objects would be black with just some reflections showing. The second part is the diffuse reflection, which is calculated from the light positions, and does not take the camera position into calculation (See Equation 1.1). It represents the light reflected from a rough surface into all directions. The Last part is specular reflection (See Equation 1.2), which represents the light reflected with the same angle in respect to normal (See Figure 1.5). This produces shiny spots on the surface where the light is reflected in general direction of the viewpoint, brightest reflection being on the point the normal is pointing towards the viewpoint [9].

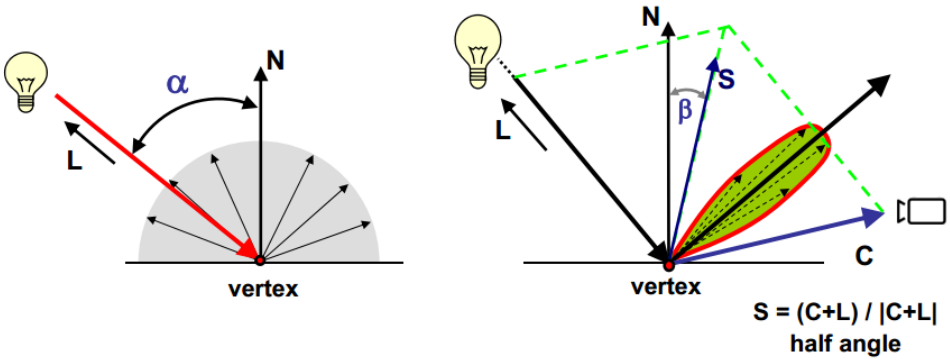

Figure 1.4: Scheme to help understand Blinn-Phong Equations 1.1 and 1.2. Adapted from [3].

L is a normalized vector from the point towards a light source
N is the normal of the surface at the point
C is a normalized vector from the point towards the camera
S is the half vector between **L** and **C**

$$d_{refl} = (max[\mathbf{L} \cdot \mathbf{N}), 0]) * d_{light} * d_{mat} \tag{1.1}$$

where: d_{refl} = reflected diffuse component
d_{light} = diffuse component of the corresponding light
d_{mat} = diffuse component of the material
$\mathbf{L}$ = vector from towards the corresponding light (Figure 1.4)
$\mathbf{N}$ = the normal of the surface (Figure 1.4)

$$s_{refl} = (max[\mathbf{S} \cdot \mathbf{N}), 0])^{sh_{mat}} * s_{light} * s_{mat} \tag{1.2}$$

where: s_{refl} = reflected specular component
s_{light} = specular component of the corresponding light
s_{mat} = specular component of the material
sh_{mat} = shininess value of the material
$\mathbf{S}$ = half vector between $\mathbf{L}$ and $\mathbf{C}$ (Figure 1.4)
$\mathbf{N}$ = the normal of the surface (Figure 1.4)

1.3 Whitted-style ray tracing

While ray casting worked for determining which objects were visible and by using local illumination model like Phong illumination we could make the image look almost photorealistic, upon looking at a more complex scenes with more objects we would quickly see that if we do not render the relations between them the image will look artificial. That is why Turner Whitted expanded the algorithm in 1979 [10] to recursively trace more rays to determine the resulting color of the pixel. The first type of ray he traced after finding the closest primary ray collision was a shadow ray. One shadow ray is generated for every light source in the scene and each one has its origin at the point of collision and a direction towards a light source in the scene. In case that the shadow ray intersects any object at a point closer than the correpsonding light, the primary collision point is in shadow from that light. On the other hand, if there is no collision, the light illuminates the point and its color and intensity are added towards the final color of the pixel. This effectively creates accurate shadows, which would not have been possible without recursive ray tracing.

Provided that the colliding object has a reflective material, another type of ray is generated. Reflection ray originates from collision point and angle between its direction $\mathbf{R}$ and surface normal $\mathbf{N}$ is the same as an angle between $\mathbf{N}$ and vector towards viewpoint $\mathbf{V}$ (see Figure 1.5 and Equation 1.3)

$$\mathbf{R} = 2(\mathbf{V} \cdot \mathbf{N})\mathbf{N} - \mathbf{V} \tag{1.3}$$

where: $\mathbf{R}$ = reflected vector (Figure 1.5)
$\mathbf{V}$ = negative of incoming light vector (Figure 1.5)
$\mathbf{N}$ = the normal of the surface (Figure 1.5)

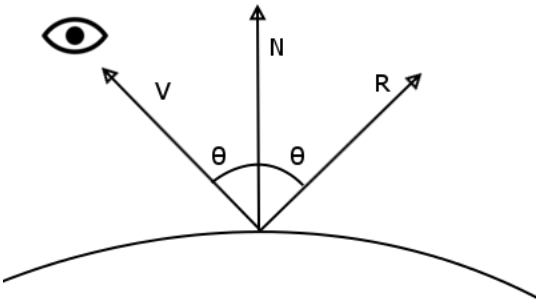

Figure 1.5: Scheme illustrating that the angle of incidence is the same as the angle of reflection.

The third type of ray introduced by Whitted was refraction ray, which is used to render transparency (see Figure 1.6). Deriving the formula for refraction vector is a bit more complex than reflection, because this time we need to know refraction indices of the scene and object through which we want to trace the ray. The calculation relies mainly on Snell's law, which describes relationship between angle of incidence and angle of refraction. For resulting formula you can see Equations 1.4, 1.5, 1.6

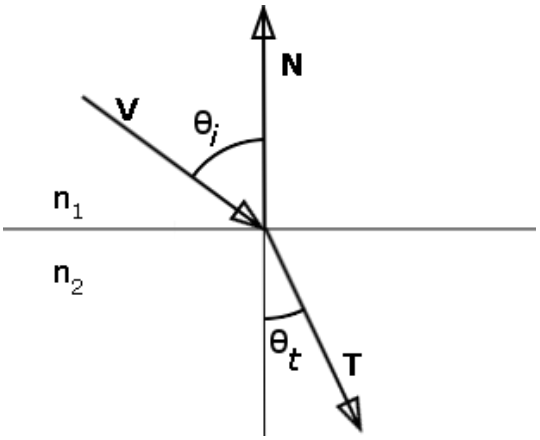

Figure 1.6: Scheme of ray refraction to help understand Equations 1.4, 1.5, 1.6.

$$\mathbf{T} = \frac{n_1}{n_2}\mathbf{V} + \left(\frac{n_1}{n_2}\cos\theta_i - \sqrt{1 - \sin^2\theta_t}\right) \tag{1.4}$$

$$\cos\theta_i = -\mathbf{V}\cdot\mathbf{N} \tag{1.5}$$

$$\sin^2\theta_t = (\frac{n_1}{n_2})^2(1-\cos^2\theta_i) \tag{1.6}$$

where: $\mathbf{T}$ = (Figure 1.6)
$\mathbf{V}$ = incoming light vector (Figure 1.6)
n_1 = refraction coefficient of the first material (Figure 1.6)
n_2 = refraction coefficient of the second material (Figure 1.6)
θ_i = angle of incidence (Figure 1.6)
θ_t = angle of refraction (Figure 1.6)

These three new ray types can bring a lot of photorealism to an image, but come at a high computation cost, where reflection and refraction rays can sometimes recursively create infinite number of new ones, which is why we always specify the recursive depth we are willing to compute for. These in combination with shadow rays can also create a lot of problems as shadow rays need to also be evaluated for every point hit by reflection and refraction rays. Quantity of lights in the scene is also a concern, as each new light brings a large number of extra shadow rays needed to be computed.

1.4 Antialiasing

Aliasing is an imprecision which often occurs during sampling process when a high frequency signal is sampled by a lower frequency. Since ray tracing is a form of sampling and more often then not we are sampling high frequency scene with lower frequency of screen pixels, aliasing can occur. It could result in the loss of some very small details on our objects or just general jaggedness of the image.

The simplest way to prevent that is to supersample the image by shooting several rays through each pixel instead of one, each offset by a small amount (Figure 1.7). However, this creates quite a significant computation overhead, as the number of rays we need to trace grows several times.

One of the possible optimizations for this method is called adaptive sampling. At first, we only trace several rays through the pixel (for example in the corners) and if the resulting colors are too different we then generate more rays to trace, thus creating more accurate antialiasing only in parts of the image that need it.

1.5 Distributed ray tracing

Using recursive ray tracing Whitted introduced to us allows us to render perfect reflections and shadows. Results gained by this method would be photorealistic assuming all real world objects were either fully reflective or nonreflective, or that every light source is a perfect point. This, however, is certainly not true and in real world most objects are partially reflective and

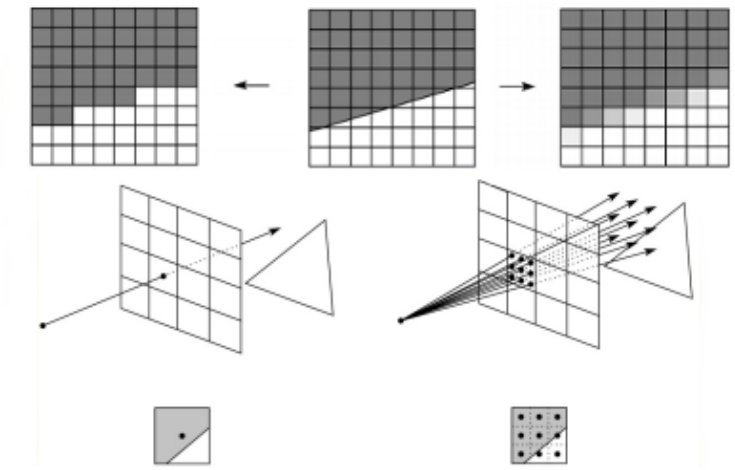

Figure 1.7: Visual illustration of antialiasing process. By tracing multiple rays through each pixel and comparing the results, we get much smoother and realistic looking edges in our image. Adapted from [4].

there is no such thing as a perfect point light source. Distributed ray tracing is yet another improvement of the ray tracing algorithm, which aims to address these issues. The way it does it is actually pretty simple and we have already talked about antialiasing, which uses the same principle. The basic idea is to generate additional rays for each ray we would have generated before, each with a slight offset depending on a few parameters and averaging the results. [11] This can be reffered to as multisampling and the specific ray we need to multisample depends on the result we want to get. The performance hit, same as with antialiasing, is again very big, but that is the cost of photorealism.

1.5.1 Gloss and Translucency

When an object is not perfectly reflective or perfectly transparent, it causes the light to be reflected or refracted in more than one direction. The way we simulate this behaviour is that we calculate the offset of rays based on reflectivity or transparency coefficient and then generate multiple random reflection or refraction rays within this offset. By averaging the resulting values we get a much more realistic result we would not have been able to get before.

1.5.2 Soft Shadows

Same as before, but this time we have to know the sizes of individual light sources and each time we would cast a shadow ray we generate multiple of them, each pointing towards a random point around the light within the spe-

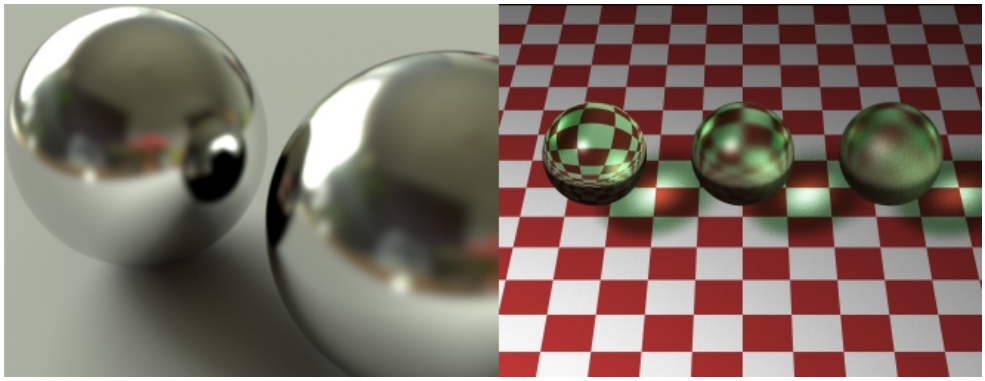

Figure 1.8: Examples showing gloss (left image), translucency (right image) and soft shadows (both images). All of these effects can be achieved by distributed ray tracing, but are very computationally demanding.

cified size. The result may range from shadows being softer near the edges to being almost invisible for small or thin objects, just like in real world.

1.5.3 Depth of field

An effect which causes objets that are too close or too far to be blurrier is in photography caused by the size of the aperture within a camera. The wider the aperture, the blurrier the photo will be. In ray tracing however, all our rays origin from a single point, which is equivalent to having a very small aperture through which only one photon could pass. To achieve depth of field we could calculate an offset and generate multiple additional rays within that offset for every pixel.

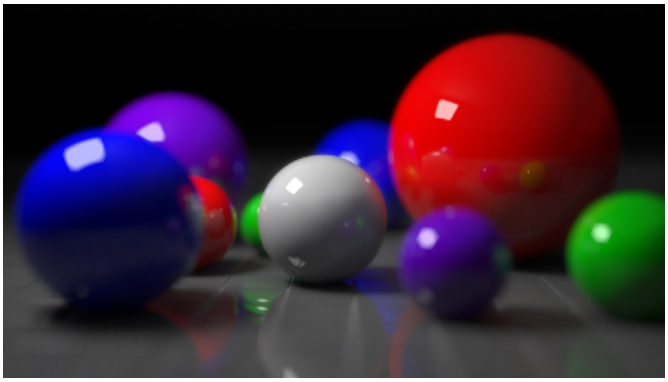

Figure 1.9: An example image showing depth of field, another effect achievable by distributed ray tracing, rendering distant images to be blurry, simulating photography. Adapted from [5].

1.6 Acceleration data structures

One of the biggest performance drawbacks of ray tracing is the need to find every ray object intersection in the scene, which naively involves testing every ray against every primitive in the scene, while in most scenes, there are considerably more primitives than the number of collisions found. Naturally, finding a way to speed up this process and filter out some unnnecessary tests became the goal of many researchers in the field. Generally, there are three main acceleration data structure types: Grid structures, Tree structures and Bounding volume hierarchies.

Chapter 2

GPU Parallelization

In the past, improvement in CPUs was mainly being accomplished by increasing the clock speed of invdividual processing unit. While transistors, the basic building blocks of processing hardware, are getting smaller year by year, the processing speeds of individual computing cores has started to plateau in early 2000s (Figure 2.1). The main reason is that by increasing clock speeds further, the heat production rises and we are not able to cool them easily enough for it to be feasible option for consumer use. Instead computing hardware started evolving in terms of parallelism, adding more cores to achieve greater performance. CPUs, however have not been developed with this idea in mind, so while they are consistently getting faster year by year, in terms of raw performance they were overshadowed by the graphics hardware, which has been built fundamentally for parallelism to support graphics pipeline. As a result, modern day GPUs are significantly more powerful than CPUs, and their computing power and memory bandwidth are improving at a faster rate then CPUs (Appendix Figure A.1).

2.1 NVIDIA CUDA

CUDA stands for Compute Unified Device Architecture and is a programming model introduced in 2007 by NVIDIA. Thanks to it we can now easily utilize GPU cores for general-purpose parallel programming. This can be very effective when dealing with alogrithms that perform the same calculation many times on different data. The downside is that it only works on NVIDIA GPUs from G8x series onwards.

2.2 Programming model

CUDA itself involves libraries that act as an extension to widely used programming languages, natively C/C++ and fortran, but third party wrappers

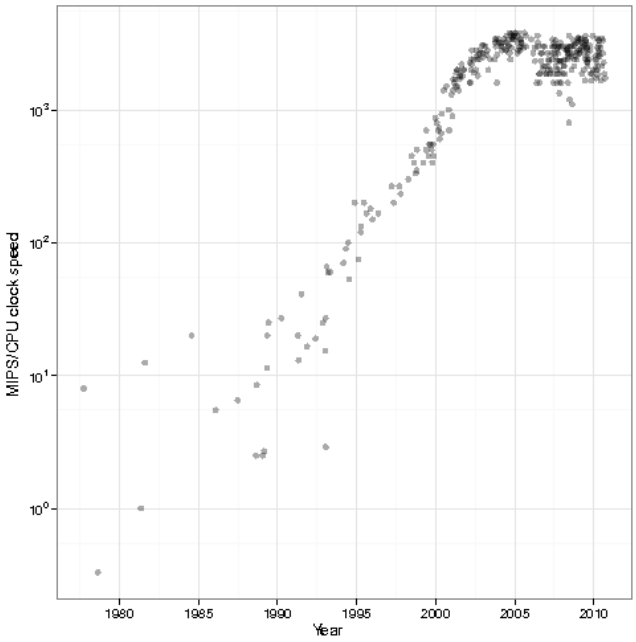

Figure 2.1: This graph shows clock speeds of individual Intel CPUs until year 2010. We can clearly see that clock speeds have plateaued at around 3.4 GHz. Adapted from [6].

exists which add support for other common languages such as Java, Python, Fortran or Lua.

A program utilizing CUDA involves running code on both CPU and GPU, for which CUDA uses terms host and device respectively, which also covers memory associated with each of them. The main responsibilities the host has are allocating memory on the device, copying data from host to device or device to host and launching kernel function, which is what we call the function designed to run in parallel on the device.

Host and device source code can be mixed in the same file. The CUDA compiler, in our case NVIDIA C Compiler (NVCC), will parse the program and split it into parts that will run on CPU and the GPU, generating code for each part separately. The device code is compiled with Cuda C Compiler (CUDACC), which generates CUDA object files. Those are then linked with CPU object files, creating an executable.

2.3 CUDA Code examples

To write a functional CUDA application, we need to understand a few main capabilities this programming model provides us. The following sections in-

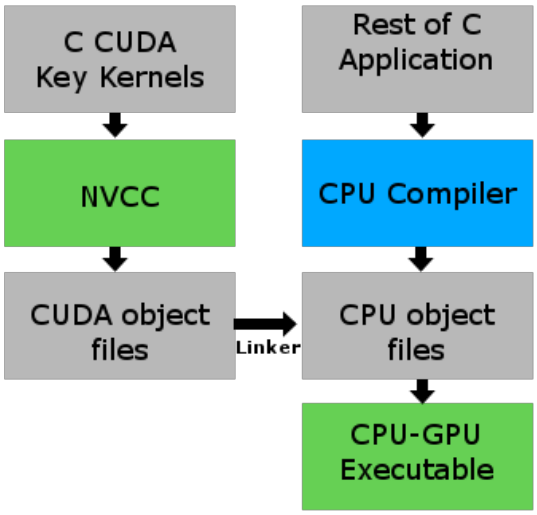

Figure 2.2: Compilation of a CUDA program using NVCC.

clude some examples of CUDA constructs in C/C++ programming languages. Altoguh CUDA provides us with many more features than the ones described below, for our purposes these should be enough.

2.3.1 Define a kernel function

Kernel is a name for our main device function meant to be run in parallel on many threads. To specify that function is a kernel we use the keyword *__global__*. The kernel has to return *void* and can have any number of parameters.

```
__global__ void exampleKernel() {...}
```

2.3.2 Specify whether a function is meant to run on host or device code

Similarly to how we defined a kernel, we can mark any function in our program as host or device function.

```
void hostFunctionA() {...}
__host__ void hostFunctionB() {...}
__device__ void deviceFunction() {...}
__host__ __device__ void sharedFunction() {...}
```

2.3.3 Allocate memory on the GPU

We do this using *cudaMalloc()* function, into which we pass these arguments:

- a reference to a pointer, which will point to device memory after allocation is complete
- size of desired allocated space in bytes

```
int* d_a;
cudaMalloc(&d_a, ARRAY_LENGTH * sizeof(int));
```

2.3.4 Move data between RAM and VRAM

As before there is a function for this called *cudaMemcpy()* and the arguments are:

- pointer to target memory
- pointer pointing to memory from which we want to copy our data
- length of data to be copied in bytes
- either *cudaMemcpyHostToDevice* or *cudaMemcpyDeviceToHost* depending on the direction we want to copy our data in.

```
cudaMemcpy(d_a, a, COPY_LENGTH, cudaMemcpyHostToDevice);
cudaMemcpy(a, d_a, COPY_LENGTH, cudaMemcpyDeviceToHost);
```

2.3.5 Launch the kernel from CPU on the GPU

Finally, the most important part is launching our kernel on multiple threads. The way parallelism works on GPU, the threads are typically launched in multiple blocks. We can launch as many blocks as we like, but the maximum number of threads per block varies on the GPU and is typically 1024. It is common to call *cudaDeviceSychronize()* after invoking the kernel to wait until GPU has finished its computations before continuing with host code.

```
exampleKernel<<<NUM_OF_BLOCKS, THREADS_PER_BLOCK>>>();
cudaDeviceSynchronize();
```

2.3.6 Working program example

This is a simple example of a program that adds two vectors and writes the result into the third one. The example features all of the above principles to showcase their use.

```
#include <iostream>
#include <cuda.h>
#define VECTOR_LENGTH 10
using namespace std;
```

```
__device__ void printIndex(int i) {
    printf("Hello, I am thread %d\n", i);
}
__global__ void vectAddKernel(int* a, int* b, int* c) {
    int i = threadIdx.x;
    printIndex(i);
    c[i] = a[i] + b[i];
}
int main() {
    // host vectors
    int a[VECTOR_LENGTH];
    int b[VECTOR_LENGTH];
    int c[VECTOR_LENGTH];
    //fill the host vectors
    for (int i = 0; i < VECTOR_LENGTH; i++) {
        a[i] = i;
        b[i] = VECTOR_LENGTH - i;
    }
    // device vectors
    int* d_a;
    int* d_b;
    int* d_c;
    // allocate space on the device
    cudaMalloc(&d_a, VECTOR_LENGTH*sizeof(int));
    cudaMalloc(&d_b, VECTOR_LENGTH*sizeof(int));
    cudaMalloc(&d_c, VECTOR_LENGTH*sizeof(int));
    // copy data from host to device
    cudaMemcpy(d_a, a, VECTOR_LENGTH*sizeof(int),
                                    cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, VECTOR_LENGTH*sizeof(int),
                                    cudaMemcpyHostToDevice);
    // launch the kernel for 1 block and
    // VECTOR_LENGTH of threads per block
    vectAddKernel<<<1,VECTOR_LENGTH>>>(d_a, d_b, d_c);
    cudaDeviceSynchronize();
    // copy the result vector back to RAM
    cudaMemcpy(c, d_c, VECTOR_LENGTH*sizeof(int),
                                    cudaMemcpyDeviceToHost);
    // write the resutls
    for (int i = 0; i < VECTOR_LENGTH; i++) {
        cout << c[i] << endl;
    }
    return 0;
}
```

The resulting vector c will have *VECTOR_LENGTH* value in every field, as expected.

Chapter 3

Implementation

3.1 Choosing the scope of my work

When I have been preparing to work on my ray tracer, I needed to decide on what will it actually do and how many features I wanted to implement. I decided to build a simple minimal viable product and build up from it. Finally I ended up with ray tracer that, on top of basic Phong shading, supports Whitted-style ray tracing, meaning it creates shadows, reflections and refractions of rays. As for the supported objects, I only ended up implementing spheres and planes, because collision detection with them is quite straightforward. I could have implemented triangles and supported common 3d models, but I decided against it, because it wouldn't help me show much more in terms of GPU parallelization, it would only make the code more cluttered, because of limitations CUDA places on polymorphism and virtual functions. There was also an option of using a thrid party framework for collision detection, but I decided against that as well, as I wanted to study the algorithm properly and understand everything I was doing. Furthermore, the aim of this thesis is to study differences between CPU and GPU computation, so I used simpler scene, because it serves the purpose just as well as a more complicated one would. I also did not implement any acceleration structures for my algorithm, as they are mainly useful when there are thousands of triangles in the scene, and not as much for the few objects I will be testing it on. Lastly, I ended up implementing multisampling of primary rays, which provides an anti-aliasing effect. I am now going to describe the structure of a program running on the CPU and later get into pitfalls of trying to get it working on the GPU.

3.2 Basic data structures

3.2.1 Vect

A *Vect* class is a 3 dimensional vector used for holding coordinates and capable of performing various operations on them. Similar type of structure is needed for all applications dealing with graphics computations in 3d space. A class with same basic interface as shown below was used in our implementation.

```
class Vect
{
public:
    double x, y, z;

    double magnitude();
    Vect normalize();
    Vect negative();
    Vect vectAdd(Vect v);
    Vect scalarMult (double scalar);
    double dotProduct(Vect v);
    Vect crossProduct(Vect v);
    ...
};
```

3.2.2 Color

This class is very similar to Vect in terms of its interface, since it holds 3 values as well and performs subset of operations vector does on them. However, it is implemented separate noneteheless, because it is logically a different structure and it also allows us to clamp its red, green and blue values between 0.0 and 1.0.

```
class Color
{
public:
    double r, g, b;
    Color add(Color);
    Color multiply(Color);
    Color multiply(double);
    ...
};
```

3.2.3 Ray

This is a very simple class that only serves as a container for two vectors, one being the origin of the ray and the other being its direction.

```
class Ray
{
```

```
public:
    Vect origin, direction;
    ...
};
```

3.2.4 Material

This class contains all information needed to calculate the surface color of an Object. The 3 colors ambient, diffuse, specular as well as shininess value are needed for Phong shading calculation, while the remaining 3 real values are needed to calculate reflection and refraction rays.

```
class Material
{
public:
    Color ambient;
    Color diffuse;
    Color specular;
    double shininess;
    double reflectiveIndex;
    double refractionAmount;
    double refractionIndex;
    ...
};
```

3.3 Scene representation

3.3.1 Object

Object is a virtual class that all other objects inherit from. It only contains a *Material* member and defines interface of 2 functions each object has to overwrite. First is *findIntersection(Ray)*, which takes a *Ray* and returns the distance at which the ray intersects the object, or -1 if no intersection was found. Second is *getNormalAt(Vect)* which returns normal vector of the object at the given point in space.

```
class Object
{
public:
    Material material;
    virtual double findIntersection(Ray ray) = 0;
    virtual Vect getNormalAt(Vect point) = 0;
    ...
};
```

3.3.2 Plane

Plane class inherits from *Object*, and in addition to declaring its own *findIntersection(Ray)* and *getNormalAt(Vect)* methods, it contains *Vect* representing the normal and *double* representing the distance of the plane along the normal, which is a regular way of reperesenting plane in analytic geometry.

```
class Plane : public Object
{
public:
    Vect normal;
    double distance;
    ...
};
```

3.3.3 Sphere

Similar to plane, only sphere is defined by its center and radius.

```
class Sphere : public Object
{
public:
    Vect center;
    double radius;
    ...
};
```

3.3.4 Light

The ray tracer supports point lights without direction, so all a *Light* needs is a position and its diffuse and specular color values which are used in shading calculation.

```
class Light
{
public:
    Vect position;
    Color diffuse;
    Color specular;
    ...
};
```

3.3.5 Scene

This is just a class grouping objects and lights together. Notably, it contains one color representing the ambient light in whole scene and a *Camera*.

```
class Scene
{
```

```
public:
    Camera camera;
    Color ambientLight;
    std::vector<Object> sceneObjects;
    std::vector<Light> sceneLights;
    ...
};
```

3.4 Raytracer

The *Raytracer* class contains our class and it is where all the computation is executed from. The ray tracing process is started by calling a public method *render(int,int,int)* and passing it image width, height and a number indicating the amount of multisampling to be done.

```
class Raytracer
{
private:
    void traceRay(Ray, int);
    void multisamplePixel(Vect,double,int);
    Color* calculatePhong(Vecct, Object*);
    ...
public:
    Scene scene;
    Color* render(int,int,int);
    ...
};
```

3.4.1 Overview of the rendering process

The *render* method is actually quite simple. First, it cycles through every pixel and calculates the direction of corresponding ray. Then, if we are using multisampling, it creates multiple rays, each still intersecting the same pixel, only in different offsets in a regular grid pattern. Then, it calls our *traceRay* function which does most of the work and returns a color. After that, it writes the color, or its fraction when using multisampling, into the corresponding pixel in our array. Lastly, it returns the array of pixel color values.

3.4.2 Tracing the ray

The process of tracing the ray is arguably the most important part of our program. The following pseudocode should give a good idea of how it works. The details of how different ray directions are calculated are already covered in Section 1.3.

```
traceRay(Ray ray, int depth) {
```

```
    for each object in scene {
        intersection = object.findIntersection(ray)
        if (intersection > -1
                        and intersection < min) {
            hitObject = currentObject
        }
    }
    if ( no intersection exists ) {
        result = black;
    } else {
        result = calculatePhong(intersectCoords, hitObject)

        if (depth < MAX_DEPTH) {
            if (hitObject.material.reflectiveIndex > 0) {
                -calculate direction of reflected ray
                reflectedColor=traceRay(Ray(intersectCoords,
                                reflDirection), depth+1)
                result += reflectedColor
                        * hitObject.material.reflectiveIndex
            }

            if (hitObject.material.refractionAmount > 0) {
                -calculate refraction index based on
                whether the ray is coming from outside or
                inside of the object
                -calculate direction of refracted ray
                refractColor=traceRay(Ray(intersectCoords,
                                refrDirection), depth+1)
                result += refractColor
                        * hitObject.material.refractionAmount
            }
        }
    }
    return result
}
```

3.4.3 Phong illumination calculation

We have already described how to calculate Phong illumination in Section 1.2, so we will not go into too much detail here. The only addition worth mentioning is that our implemenation also calculates shade by creating rays towards every light in the scene and testing if they intersect any object on the way to the light. It then multiplies the color by the ratio based on the number of lights illuminating the surface.

3.5 GPU implementation

The principles and constructs described above were meant to be run on CPU and needed to be changed to be feasible for GPU computation. While the main principle remained the same, there were some major areas where I had to modify them.

3.5.1 Recursion

Using recursion in CUDA code usually brings some difficulties. While first versions of CUDA did not support recursion at all, CUDA 3.1 brought dynamic parallelism supprot for devices with compute capability 2 [12]. This allowed to call child CUDA kernel from parent kernel and optionally synchronize on the completion of that child kernel. It also allowed for the kernel to call itself, thus providing recursion support. Instead, I decided to rewrite my *traceRay* method to work without recursion. I found that every traced ray contributed to the resulting color of the pixel by flat amount, only needing to know the weight to calculate how much to add to the color. I added a new class *Task* to group all the necessary information to trace any given ray and add the result to the corresponding pixel. The drawback of this approach is that I have to create all the primary tasks beforehand, as well as collect newly created tasks after each wave of GPU calculations. The advantage of this method is that instead of recursively calling *traceRay*, all we do is create a new task and add it to the stack. We do not have to wait until the task is calculated, because we already gave it enough information to be able to add the result correcltly to the right pixel.

```
class Task
{
public:
    int x;
    int y;
    double weight;
    Ray ray;
    int depth;
    ...
};
```

3.5.2 Polymorphism

As of now, there is no support for polymorphism in CUDA, which meant that the object structure of my raytracer needed to be changed. Before there was one virtual *Object* class, from which objects inherited and overwrote *findIntersection* and *getNormalAt* functions. To prevent modifying the code too much, I created a simple wrapper *CommonObject* that contained both a plane and a sphere, but acted as only one of them. This meant that our objects required

more memory than before, but since I mainly used the program to render simple scene with smaller number of objects, this did not present too much of a problem.

```
class CommonObject
{
private:
    static const int PLANE_IDENT = 0;
    static const int SPHERE_IDENT = 1;
public:
    Plane plane;
    Sphere sphere;
    int identifier;
    ...
    __host__ __device__ double findIntersection(Ray ray);
    __host__ __device__ Vect getNormalAt(Vect point);
    __host__ __device__ Material getMaterial();
    ...
};
```

3.5.3 STL vector and queue

Formerly, my program used *vector* and *queue* from the C++ Standard Template Library for various data management tasks. However, when I tried to call any function on them from device code, the compiler immediately informed me that those functions are not marked as device functions, thus it was unable to execute them. As I could not change the declaration of STL container functions, I decided to make my own container with the same interface (the part of it I was using).

3.5.4 Pointers

It is important to realize that all pointers I used on CPU pointed to host memory. In order for GPU to work with them correctly, I would have to move the corresponding data from host to device memory and supply device functions with correct device pointers. Some of my classes used pointers to data to save memory in case there was no data currently attached to it. After thinking about it I proceeded to remove most member pointers of my classes, as the memory tradeoff was much more acceptable than having to copy small blocks of data individually to the GPU. This allowed me to pass most of the data i needed to get to device as function arguments.

3.5.5 Memory management and overall flow of the render method

While the basic *render* method was already described above in Section 3.4.1, there are quite a few changes that were made when rewriting the raytracer

to run on GPU. Here is the new overview describing the steps we go through when rendering an image:

1. Create an array of *Color* to represent colors of every pixel
2. Calculate the direction of each primary ray, then create and add the corresponding *Task* to our tasks queue. When multisampling is enabled, generate multiple rays for each pixel. I use a uniform square grid to distribute rays within the pixel, which means that the number of rays generated is always a square number.
3. Create host and device pointers and allocate space on host and device memory for:
   - Batch of tasks to be sent to GPU
   - Batch of *Color* results that the GPU will write to
   - New tasks that the traced rays will generate
4. We will proceed to iterate through the following operations until our queue of tasks is empty.
   a) Copy the previously created fields to the GPU
   b) Start the kernel on the GPU. We are able to launch multiple blocks, each containing a set number of threads. The maximum number of threads depends on the GPU. Both the number of blocks and threads per block are variables in our program that we will be able to tweak and observe different results.
   c) Copy the results and new tasks back from the GPU.
   d) Merge the fresh results with the main pixel colors.
   e) Merge the new tasks with the remaining tasks.

3.5.6 CUDA Unified memory

CUDA 6 introduced a new breakthrough in memory management called Unified memory. It allows us to create a shared pool of memory, that can be accessed from both CPU and GPU. It is accessible by a single pointer which is same in both the host and device code. It accomplishes this by automatically moving data between CPU and GPU so the programmer does not have to worry about it [13]. It could be useful in our raytracer because we would not have to copy all our tasks and result colors between CPU and GPU and we could operate on the shared data in unified memory. However when I tried to utilize this principle I found that the overhead that comes with access and writing to the unified memory is actually making the overall render time higher than what it was before. There is a possibility that unified memory could be efficiently utilized in our program, however I decided to revert back to copying the data manually before and after launching each batch of threads.

Chapter 4

Results

Our raytracer has a number of different parameters that determine its performance, including:

- CPU or GPU computation, as well as the specific GPU we use
- Target resolution
- Amount of multisampling
- Scene complexity
- Depth of recursive tracing of rays
- Number of blocks and number of threads per block

This chapter will deal with running the renderer on different configurations, observing the results and trying to find analyze them. The following notations determine which configuration was used for the test.

4.1 Available testing hardware

This section includes tests performed on three different hardware configurations.

- **CPU** notation means that the test was measured solely on the CPU, specifically *Intel Core i7-3610QM CPU @ 2.30GHz*.
- **GPU1** notation marks that the test utilized *NVIDIA GeForce GTX 660M* GPU.
- **GPU2** means that the test was performed on a different machine altogether containing *NVIDIA GeForce GTX 770* GPU, supported by *Intel Core i5-4670K CPU @3.4GHz*.

4.2 Resolution

Because every pixel is computed indepenedtly of every other pixel, it is safe to assume that relationship between render time and number of pixels will be linear. The tests performed more or less confirm this (See Figure 4.1). The results can be found in Appendix Table A.1.

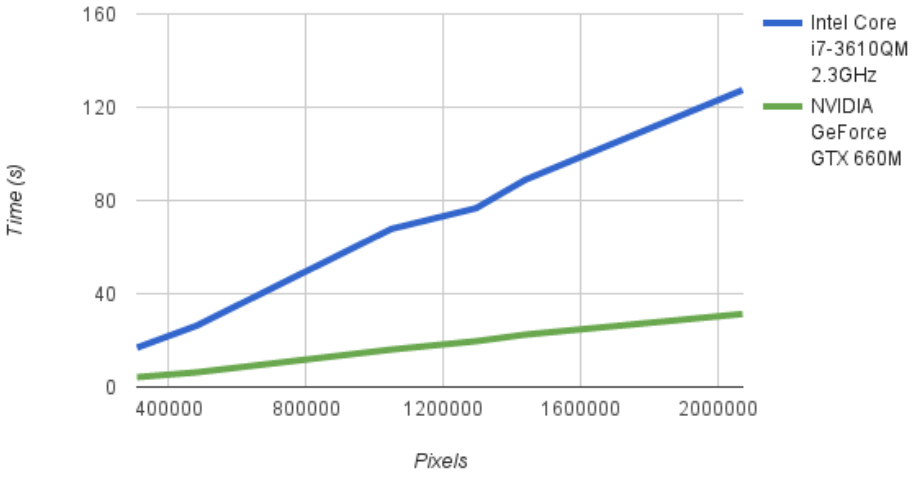

Figure 4.1: Chart demonstrating linear relationship between computation time and the number of pixels needed to be rendered.

The table of results as well as the configuration for this test can be found in Appendix Table A.1.

4.3 Multisampling

Multisampling removes the aliasing artifacts from final image by tracing more rays in slightly different directions for each pixel. Effectively, all it does is render the image in larger resolution and downscale the results. That is why it is also safe to assume that the rendering time should be more or less linear with the amount of rays multisampled for each pixel. The tests performed also confirm this and can be found in Appendix Table A.2

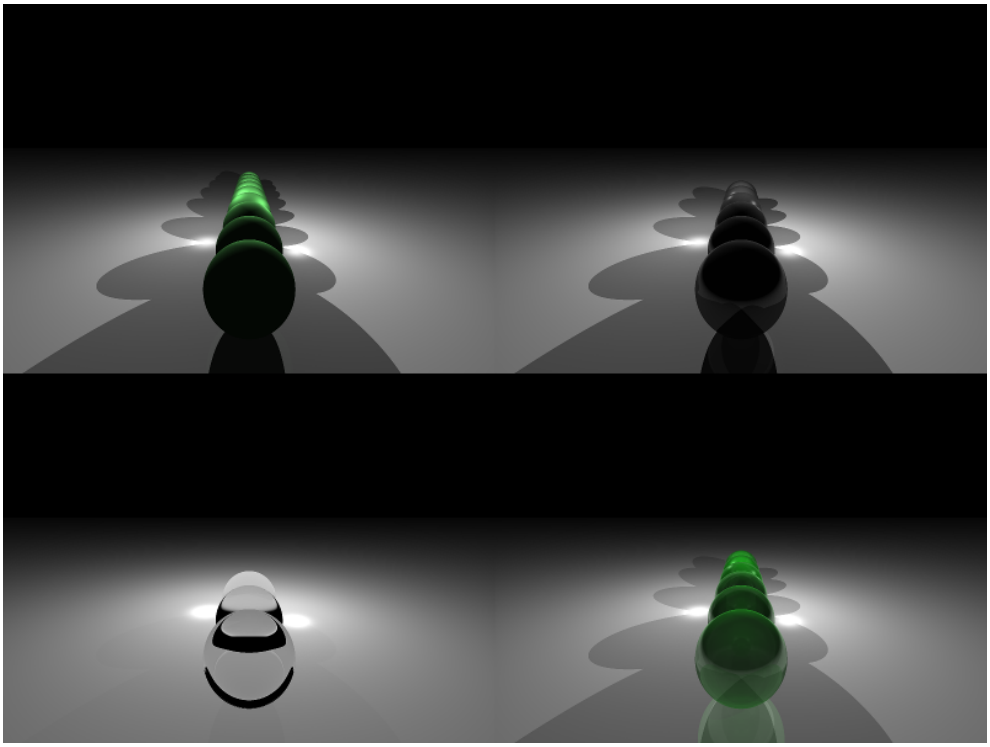

Figure 4.2: Example images generated from 4 different scene tests. We tried changing the number of spheres, as well as their material: plastic (top left), chrome (top right), glass (bottom left) and emerald (bottom right).

4.4 Scene Complexity

The goal of this test is to study the effect of different scene configurations on the resulting computation time. We performed tests on 16 different scene configurations. Our base scene contained one reflective plane that served as floor. We then created 1, 3, 6 or 9 spheres in a row in front of the camera. Each sphere had the same material and we measured the results based on the combination of the number of spheres and which material we set to it (See Figure 4.2). Each material was different from the others. The *plastic* was a simple material that wasnt neither reflective nor refractive. *Chrome* was only reflective and *glass* was only refractive. Last material, *emerald* was both reflective and refractive.
The observed results summarized in Table 4.1 show us that there were not any huge differences between *plastic*, *chrome* and *glass* material, while the render time was a lot longer when we used *emerald* material (See Figure 4.3). There is a noticeable increase in time when we are calculating *glass*, as we would expect, but the difference between *plastic* and *chrome* calculation is very small. Expected result would be that *chrome* would take some time longer, same as

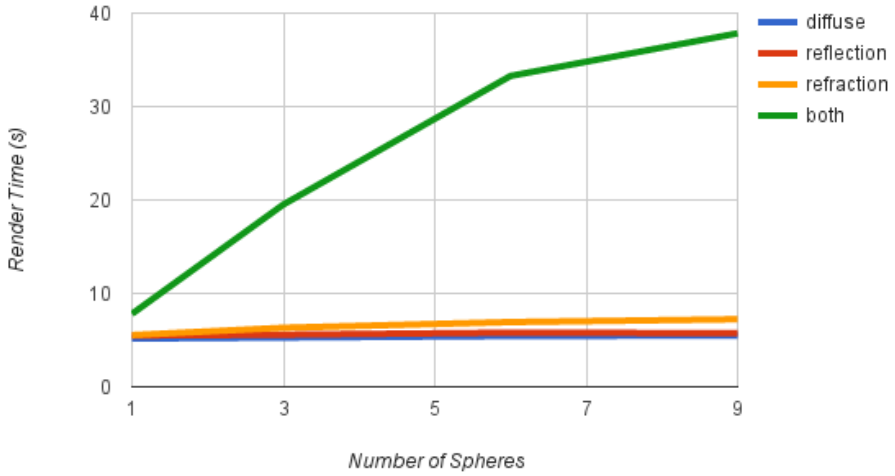

Figure 4.3: Graph showing the data from the Table 4.1.

	1 sphere	3 spheres	6 spheres	9 spheres
plastic	5.163	5.274	5.435	5.489
chrome	5.487	5.581	5.799	5.742
glass	5.536	6.319	6.931	7.232
emerald	7.811	19.518	33.278	37.838

Table 4.1: Results of the tests performed on **GPU1** showing differences in render time with different scene configurations

The *plastic* row shows results gained when all the spheres in the scene had simple material without any reflections or refractions
The *chrome* row shows results gained when all the spheres had a reflective material. Similarly, in *glass* row all the spheres had glassy refractive material. In the *emerald* row we set the material to be both reflective and refractive.
All of the tests were rendering an 800x600 image and utilized 4x multisampling.

it is with *glass*. The explanation lies in the scene we have chosen for this test. We have laid the spheres in front of the camera in such a way, that no reflections between spheres are visible and the only additional reflection the *chrome* test has to calculate is the ground reflected on the first sphere. If we were to set the scene up differently, for example laid the spheres out in a line perpendicular to the camera direction, there would be a noticeable difference in render times, as every sphere would reflect its neighbouring shperes.
Another interesting observation is the jump in render time when we set the material to *emerald*. This can be explained when we take a look at the number

of rays we are generating. In the *glass* test, we shoot a ray into the first sphere and it is refracted in the corresponding direction. It could then exit the sphere and hit the second sphere to be refracted again, but the direction of the ray always remains more or less away from the camera. In the *emerald* test, however when a refracted ray exits the first sphere and hits the second one, it could be both reflected back at the first one and refracted forward. There is a good chance that the reflected ray now travels "back to the camera". If it hits the first ray again, another two rays are generated. Because the rays now have the potential to bounce between the spheres, the number of potential rays reaches geometric amounts (it could also be infinite, if the scene was set up correctly). Because of this, the computation time now depends on the depth into which we are willing to trace the ray. The depth parameter was set to 10 in these tests, which is way more than the potential "heritage" of most of the rays traced in the first 3 tests.

4.5 Depth parameter

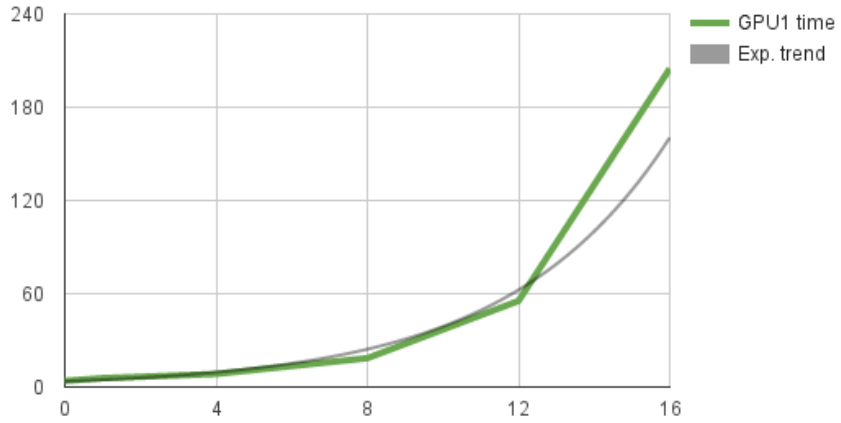

Figure 4.4: Graph showing the performance impact *depth* parameter has. We see it can significantly affect the render times. Based on data from Table 4.2

The depth test studies how does maximum allowed *depth* affect the render time. The *depth* parameter specifies the number of ray "generations" we are willing to trace. For example, when *depth* is set to 0, even if we hit a reflective or a refractive material, we are not alowed to create new rays to calculate reflction. If *depth* is 1, we can create them, however if those new rays hit

another reflective or refractive material, they will not spawn new rays.
The scene we used for this test is similar to the *emerald* tests we did when we were testing scene complexity. It contains 5 spheres in a row in front of the camera with material set to *emerald*, because this setup showed potential to spawn a huge number of new rays.
The results shown in Table 4.2 show that if the scene is generating enough new rays, increasing the *depth* parameter can significantly impact the performance. The growth in performance depends on the rendered scene as well as the number of pontential rays spawned on impact. In our raytracer, each ray can only generate up to two new rays, but if we were to implement distributed ray tracing, the performance growth would be much more severe.

Depth	GPU1 time	GPU2 time
0	3.954	2.852
1	5.697	4.123
4	8.371	6.001
8	18.541	12.647
12	55.378	36.509
16	204.865	132.014

Table 4.2: Result times from the depth test. Measured on both **GPU1** and **GPU2** configurations.

4.6 Threads per block and number of blocks

Another parameters that can impact the performance are the number of blocks we launch in our rendering cycle and the number of threads one block contains. The number of threads is specific to the GPU and in our case it is 1024, but the number of blocks can be as high as we want.
In our results (see Table 4.6 and Appendix Table A), however, we found that the times start to plateau and even rise after a certain number of blocks and higher numbers do not necessarily mean better speeds. The number of threads per block seems to behave similarly, again showing us that higher numbers could sometimes get worse results. This could be caused by the fact that cache coherence might be better when using certain combinations, or that copying memory back and forth between CPU and GPU might be more effective in certain sizes and intervals.

	1	5	25	50	100
32	254.208	66.865	25.438	21.429	19.634
128	81.266	30.418	17.572	16.596	15.736
256	47.954	24.252	16.913	15.995	15.353
512	32.227	19.219	16.211	15.822	15.189
1024	27.593	17.926	15.586	16.037	15.845

Table 4.3: **GPU1** Render times for different block and thread settings. There seems to be a consistent improvement when going from lower values to higher, but it seems to plateau at a certain point.

Rows indicate the number of threads per block
Columns indicate the number of blocks launched

4.7 Comparison of available hardware configurations

We have already made a good number of tests to determine differences in performance between our different configurations. When we look at the result tables both in this section and in the Appendix, we can clearly see that performance differences seem to be mostly the same. We can determine that running on **GPU1** configuration is approximately 4 times faster than using only **CPU**. The **GPU2** and **GPU1** are a little bit more variable at some places, but **GPU2** seems to be 1.3 to 1.6 times faster than the **GPU1**.

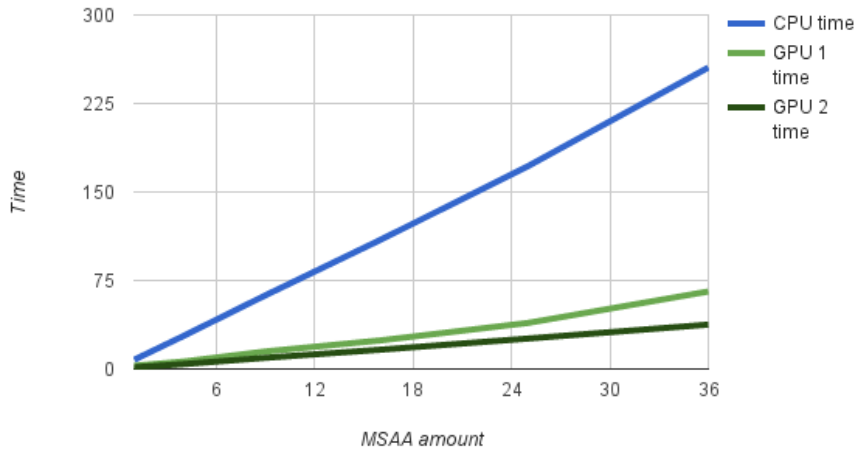

Figure 4.5: Graph based on our multisampling tests, comparing result times on all 3 hardware configurations

Conclusion

The goal of this thesis was to study the area of ray tracing, implement my own raytracer, parallelize it on the GPU and analyze the results based on different configurations.
I believe I have accomplished all these tasks, some better than others. I have studied and described the basics of ray tracing techniques and even delved into some more advanced methods.
I have implemented the raytracer using using C++ language and utilizing NVIDIA CUDA, which has a variety of features that allows it to produce a good array of results, but it has its limitations. One of them is the fact that it does not support triangles and standard triangle mesh model structures, which is also why it does not use any of the acceleration structures, which are common for advanced ray tracing renderers. It is suitable for displaying a small number of objects with different kinds of materials, including reflective and refractive ones, but is not designed for big object structures. It is, however, capable of showing the benefits of GPU-based parallelization. My GPU implementations manage to run a few times faster than the CPU ones, depending on the GPU it is using.
I also believe that I analyzed the results thoroughly, comparing different configurations and tweaking with them to reach effective speeds.

Area for future improvements

As I have already mentioned, the raytracer could be improved to support large number of objects and triangle model structures utilizing some acceleration structure to make searching for intersection faster. It does not implement distributed ray tracing, meaning it does not produce soft shadows and matte reflections and refractions. The GPU parallelization could surely be improved as well, because I know there are existing solutions that are able to render video frames in real time.

Bibliography

[1] Objects with and without illumination and shading effects. In: *Illumination and Shading (Introduction to Computer Graphics Using Java 2D and 3D) Part 1* [online]. [image]. The-Crankshaft Publishing. n.d. [viewed 7 May 2015]. Available from: http://what-when-how.com/introduction-to-computer-graphics-using-java-2d-and-3d/illumination-and-shading-introduction-to-computer-graphics-using-java-2d-and-3d-part-1/

[2] Smith, B. Illustration of the Phong Reflection Model [online]. [image]. Wikimedia Commons. 2006. [viewed 7 May 2015]. Available from: http://commons.wikimedia.org/wiki/File:Phong_components_version_4.png

[3] Felkel, P. Barva, světlo, materiály v počítačové grafice [Color, light, materials in computer graphics] [lecture]. Prague: Czech Technical University. October 2014.

[4] Fussell, D. Anti-aliased and accelerated ray tracing. In: *Computer Graphics* [online]. [PDF lecture]. University of Texas at Austin. Fall 2010. Available from: http://www.cs.utexas.edu/users/fussell/courses/cs384g/lectures/lecture10-Aa_and_accel_raytracing.pdf

[5] Mimigu. Balls Render [online]. [image]. Wikimedia Commons. 2009. [viewed 7 May 2015]. Available from: http://commons.wikimedia.org/wiki/File:BallsRender.png

[6] Gillespie, C. Intel CPU clock speed. In: *CPU And GPU Trends Over Time* [online]. [image]. 2011. Available from: https://csgillespie.wordpress.com/2011/01/25/cpu-and-gpu-trends-over-time/

[7] Žára, J.; Beneš, B.; Sochor, J.; et al. *Moderní počítačová grafika [Modern computer graphics]*. Brno: Computer Press, second edition, 2010, ISBN 80-251-0454-0, 413-435 pp.

[8] Phong, B. T. Illumination for Computer Generated Pictures. *Commun. ACM*, volume 18, no. 6, June 1975: pp. 311–317, ISSN 0001-0782. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.330.4718&rep=rep1&type=pdf

[9] Blinn, J. F. Models of Light Reflection for Computer Synthesized Pictures. *SIGGRAPH Comput. Graph.*, volume 11, no. 2, July 1977: pp. 192–198, ISSN 0097-8930. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.131.7741&rep=rep1&type=pdf

[10] Whitted, T. An Improved Illumination Model for Shaded Display. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA: ACM, 2005. Available from: http://doi.acm.org/10.1145/1198555.1198743

[11] Cook, R. L.; Porter, T.; Carpenter, L. Distributed Ray Tracing. *SIGGRAPH Comput. Graph.*, volume 18, no. 3, Jan. 1984: pp. 137–145, ISSN 0097-8930. Available from: http://artis.inrialpes.fr/Enseignement/TRSA/CookDistributed84.pdf

[12] Dynamic Parallelism in CUDA [online]. [PDF document]. Nvidia Corporation. 2012. Available from: http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief_Dynamic_Parallelism_in_CUDA.pdf

[13] Harris, M. Unified Memory in CUDA 6 [online]. [article]. Nvidia Corporation. November 2013. Available from: http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6

[14] Floating-Point Operations per Second for the CPU and GPU. In: *CUDA C Programming Guide* [online]. [image]. Nvidia Corporation. 2015. [viewed 7 May 2015]. Available from: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

Appendix A

Additional charts and tables

This section contains additional tables and charts that were not considered important enough to be included in the main thesis.

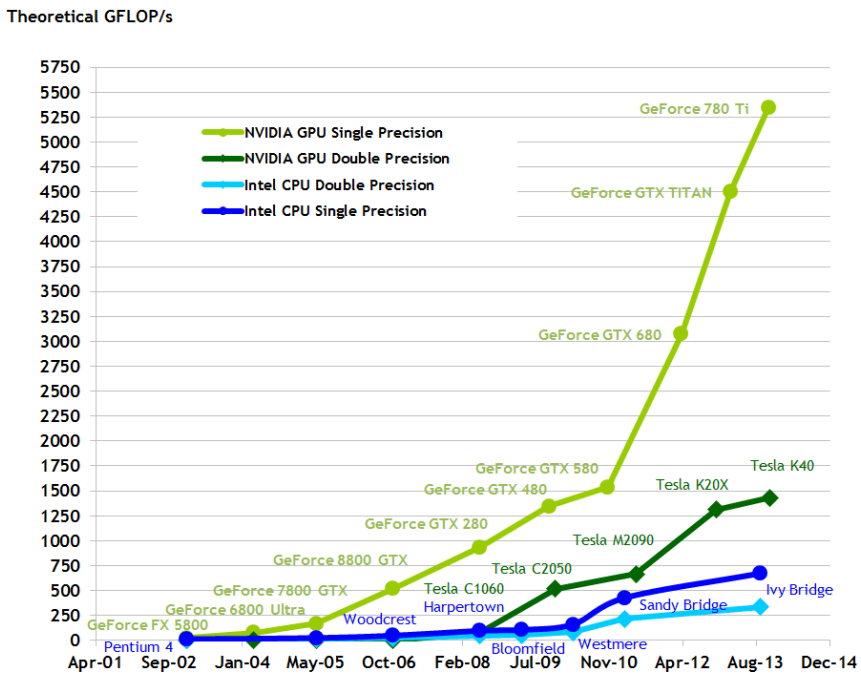

Figure A.1: Comparison of various GPUs and CPUs over the years

Adapted from [14]

Resolution	Pixel Count	CPU (s)	GPU1 (s)	CPU pixels/s	GPU1 pixels/s
640x480	307200	16.876	4.214	18203.36573	72899.85762
800x600	480000	26.235	6.273	18296.16924	76518.41224
1366x768	1049088	67.786	16.057	15476.47007	65335.2432
1440x900	1296000	76.673	19.635	16902.9515	66004.58365
1600x900	1440000	88.894	22.468	16199.06855	64091.15186
1920x1080	2073600	127.366	31.316	16280.64005	66215.35317

Table A.1: The results of testing how resolution affects render time on CPU and GPU

This test was performed on **CPU** and **GPU1** hardware configurations.
The scene used in this test contains 1 plane 4 spheres and 2 lights, with all types of materials including reflections, refractions and a combination of them. Each pixel was also multisampled 4 times.

MSAA	CPU (s)	GPU1 (s)	GPU2 (s)	CPU t/n	GPU1 t/n	GPU2 t/n
1	7.808	2.765	1.382	7.808	2.765	1.382
4	27.954	6.287	4.117	6.9885	1.5718	1.0293
8	62.424	14.838	9.453	6.936	1.6487	1.0503
16	109.367	24.206	16.323	6.8354	1.5129	1.0202
25	171.842	39.009	25.783	6.8737	1.56	1.0313
36	255.47	65.663	37.611	7.0964	1.824	1.0448

Table A.2: Results showing inrease of time with increasing multisampling amount and comparing results on different GPUs

We performed this test on all available configurations (**CPU**, **GPU1** and **GPU2**).
Columns are from left to right: amount of multismapling for each pixel, times for all three configurations we tested on. Coefficients calculated by dividing the time spent by the multisampling amount. They stay more or less the same, showing that the relationship between multisampling amount and render time is linear.
The scene used for this test was the same as in Table A.1, rendered at 800x600 resolution

	1 sphere	3 spheres	6 spheres	9 spheres
diffuse	3.932	4.08	4.181	4.214
reflection	4.231	4.318	4.278	4.378
refraction	4.151	4.747	4.997	5.151
both	7.412	17.37	22.605	24.471

Table A.3: Results of the tests performed on **GPU2** showing differences in render time with different scene configurations

The configuration is the same as 4.1, only this time the tests were performed on **GPU2**.

	1	5	25	50	100
32	199.629	50.923	17.74	13.542	11.494
128	60.844	20.381	11.393	10.8	11.031
256	38.173	14.892	11.046	11.02	11.024
512	25.899	12.23	11.158	11.047	10.791

Table A.4: **GPU2** Render times for different block and thread settings

Rows indicate the number of threads per block
Columns indicate the number of blocks launched

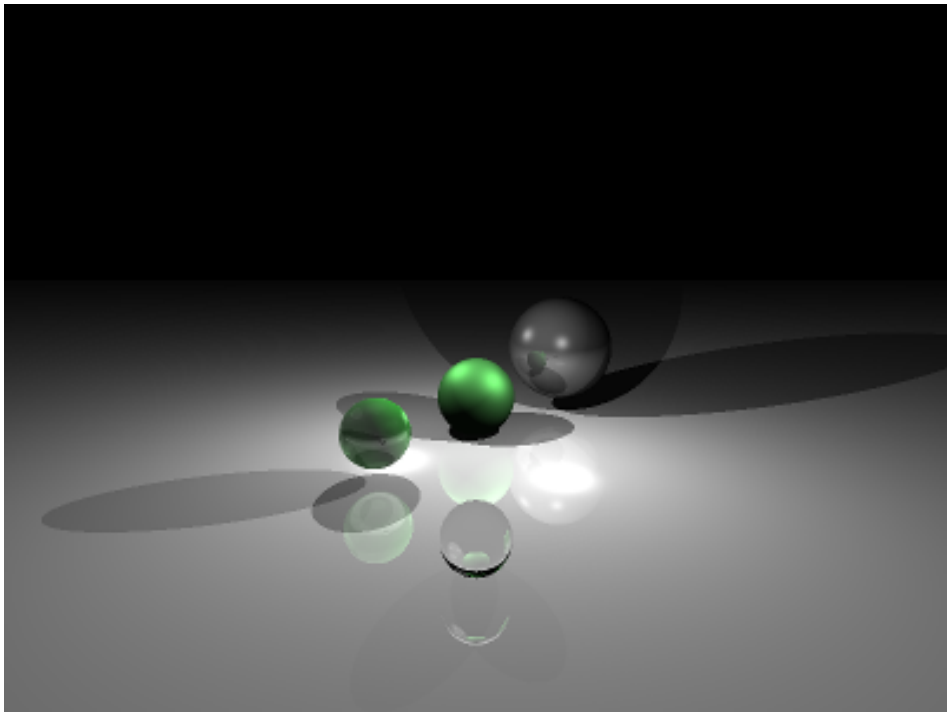

Figure A.2: An example result from our raytracer.

Appendix B

Acronyms

CPU Central Processing Unit

GPU Graphics Processing Unit

CUDA Compute Unified Device Architecture

NVCC Nvidia C Compiler

CUDACC CUDA C Compiler

RAM Random-access memory

VRAM video RAM - GPU memory

Appendix C

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `bin` ..... the directory with executables
- `results` ..... the directory of results
  - `images` ..... sample result images
  - `tests` ..... spreadsheets of data results from performed tests
- `src` ..... the directory of source codes
  - `raytracer` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format