České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Ivan Platonov**

Studijní program: Softwarové technologie a management
Obor: Web a multimedia

Název tématu: **Interaktivní vizualizace černých děr**

Pokyny pro vypracování:

Vytvořte interaktivní vizualizaci černých děr. Prostudujte možnosti knihovny vyvíjené v Astronomickém ústavu Akademie věd ČR pro vizualizaci černých děr a navrhněte její zakomponování do interaktivního systému, kdy budete s její pomocí vizualizovat změnu vstupních parametrů v reakci na uživatelský vstup v reálném čase. Vstupem systému bude nějaká podoba bezkontaktního ovladače, jako je například Kinect nebo Leap motion.

Vyberte vhodnou skupinu modifikovatelných parametrů knihovny, vyberte vhodný bezkontaktní ovladač a navrhněte mapování parametrů na uživatelský vstup. Vhodně nastavte škálu a limity vstupních parametrů. Realizujte finální systém tak, aby byl uživatelsky přívětivý bez nutnosti servisních zásahů. Zhodnoťte nároky systému, rychlost odezvy a uživatelskou přívětivost.

Použité technologie, C/C++, CUDA.

Seznam odborné literatury:

Marius Shekow, Leif Oppermann. On Maximum Geometric Finger-Tip Recognition Distance Using Depth Sensors. Communication papers proceedings of Winter School of Computer Graphics (WSCG) 2014.

Depth Sensor Shootout - https://stimulant.com/depth-sensor-shootout-2/, visited 4.1. 2016.

Vedoucí: Ing. David Sedláček, Ph.D.

Platnost zadání: do konce zimního semestru 2017/2018

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
L.S.
-4-

V Praze dne 29. 2. 2016

Bachelor's Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Black Holes Interactive Visualization

Ivan Platonov

Study Programme: Software Technologies and Management
Study Field: Web and Multimedia

May 2016
Supervisor: Ing. David Sedláček, Ph.D.

Acknowledgement / Declaration

I would like to thank Ing. David Sedláček, Ph.D. for his help and guidance, also I would like to thank Ing. Jaroslav Sloup for his invaluable advices.

I declare that I worked out the presented thesis independently and I quoted all used sources of information in accord with Methodical instructions about ethical principles for writing academic thesis.

....
Ivan Platonov
In Prague 26. 05. 2016

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

....
Ivan Platonov
V Praze dne 26. 05. 2016

Abstrakt / Abstract

Tato bakalářská práce se věnuje vizualizaci velmi vzácných a málo prozkoumaných objektů vesmíru, kterými jsou černé díry. Tato práce také popisuje matematické základy černých děr, Kerrův časoprostor a metody sledování paprsků v Kerrově časoprostoru. Pro přidání interaktivity se v této práci používá Leap Motion. Pro příznivý výkon byla použita technologie NVIDIA CUDA.

Klíčová slova: černé díry; vizualizace; NVIDIA CUDA; OpenGL; Leap Motion; interaktivita.

Překlad titulu: Interaktivní vizualizace cerných děr

This bachelor thesis presents a complete process of creating a real-time visualization of some of the most unexplored objects in the universe – black holes. It describes the basics of black holes nature as well as Kerr spacetime and ray tracing in this spacetime. As the main source of interactivity was used Leap Motion controller. For better performance and more favorable interactivity NVIDIA CUDA technology was used.

Keywords: black holes; visualization; NVIDIA CUDA; OpenGL; Leap Motion; interactivity.

Contents

Tables / Figures

Chapter 1
Introduction

A black hole is a region of spacetime where gravity is so strong that even photons of light can not get out.

In most cases black holes are expected to form at the final stage of the evolution of some very massive stars. At the end of star's life cycle it collapses under the force of their own gravity producing a black hole.

Black holes generally are invisible objects and we can visualize only the special ones. Those black holes, which will continue to grow absorbing it's surroundings because of its strong gravity. Matter falling onto a black hole and heated by friction, will create a thin disk of visible light which is called accretion disk.[1] [2]

1.1 Visualization

The gist of the project is visualization of the black hole's mathematical model written by Michal Bursa [3]. We will create physically accurate and good looking visualization of the black hole that is based on that model. We are not going to use classic 3D polygonal modeling paradigm, because our goal is to create visualization of pure mathematical model. To be exact we need to visualize geodesic equations that are part of the ray tracing inside the Kerr spacetime. To reach this goal we will use interesting techniques and technologies.

1.2 Interaction

Static picture of a black hole looks great; however, it would be better if we add to it some interaction. Within the confines of the our project we will create basic camera movement in the space around the black hole. To control the camera the application will use common input sources, such as keyboard and mouse, but also we will connect hand tracking device to make the application even more entertaining.

1.3 Performance

If we speak about the time to draw mathematically accurate model of a black hole we need a lot of performance. So in order to provide the best user experience at interaction with a black hole, we need something more powerful and faster than CPU. We will use NVIDIA-based GPU powered by CUDA technology. This will help us significantly (up to 10 times) increase the performance and make the application even more fluent. After the implementation we can compare both ways of frame creation and make a final decision, that will tell us which technology better fits our goals.

Chapter 2
Analyses

2.1 Theoretical background

In this section we will take a closer look to the theoretical part of the application and try to schematically describe the basics of the black holes. How it affects its surroundings and what we need to do to create our visualization.

2.1.1 Basic knowledges

A black hole is an extreme astronomical object that is the final stage of the evolution of some stars. Its gravitational field is so intense that it bends the path of light rays, giving rise to visible distortions.

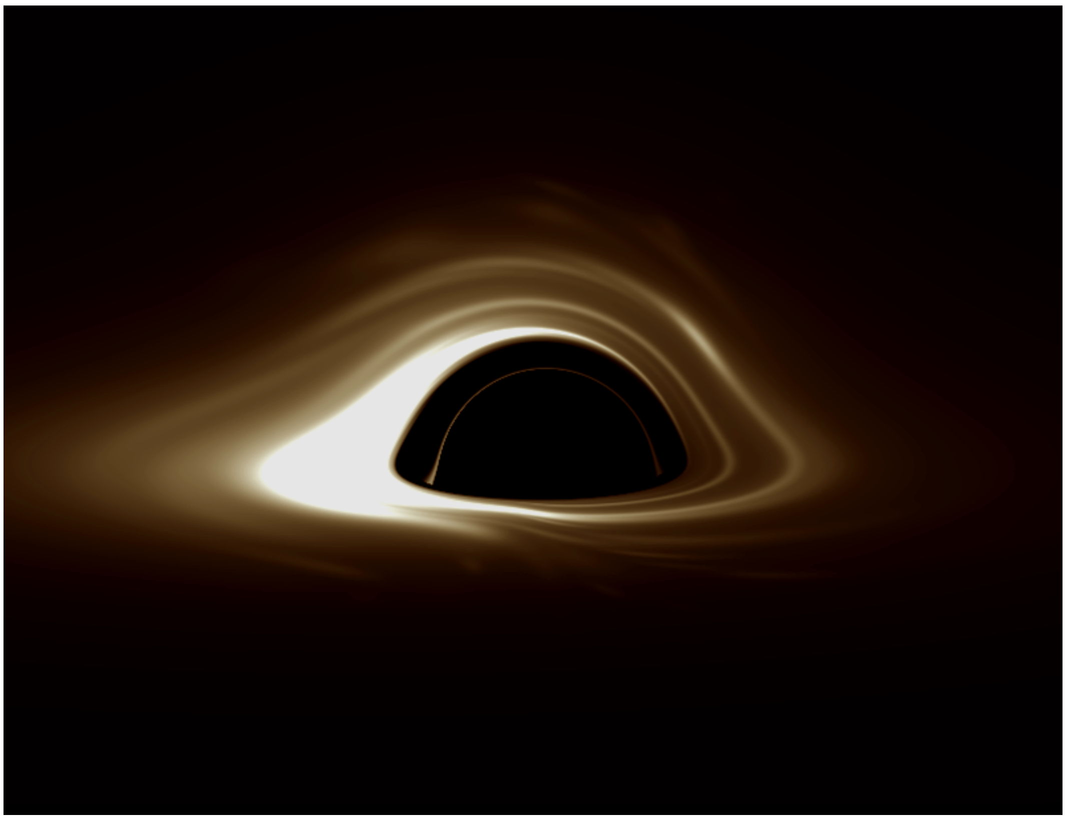

Figure 2.1. The black hole with an accretion disk around it.

The most prominent feature is the black hole's event horizon. The event horizon is the so-called "surface of no return", a membrane that can only be crossed by ingoing objects. Since nothing, including light, can escape the event horizon, its image appears pitch black. [2]

Material, such as gas, dust that has come close to a black hole but not quite fallen into it, forms a flattened band of spinning matter around the event horizon called the accretion disk. Although no-one has ever actually seen a black hole or even its event horizon, this accretion disk can be seen, because the spinning particles are accelerated to tremendous speeds by the huge gravity of the black hole, releasing their potential energy in a form of heat and powerful x-rays out into the universe.[4]

2.1.2 Visualization principles

To understand what we are going to visualize lets make a short overview of the black hole. As we already know the black hole is a region of spacetime and it works like a gravity lens.

For better understanding lets take a look to the Figure 2.2. Red arrow represents light and in the middle of the plane is a black hole. Because of the black hole's huge mass the light is "falling" inside the black hole simply as water falls inside a hole.

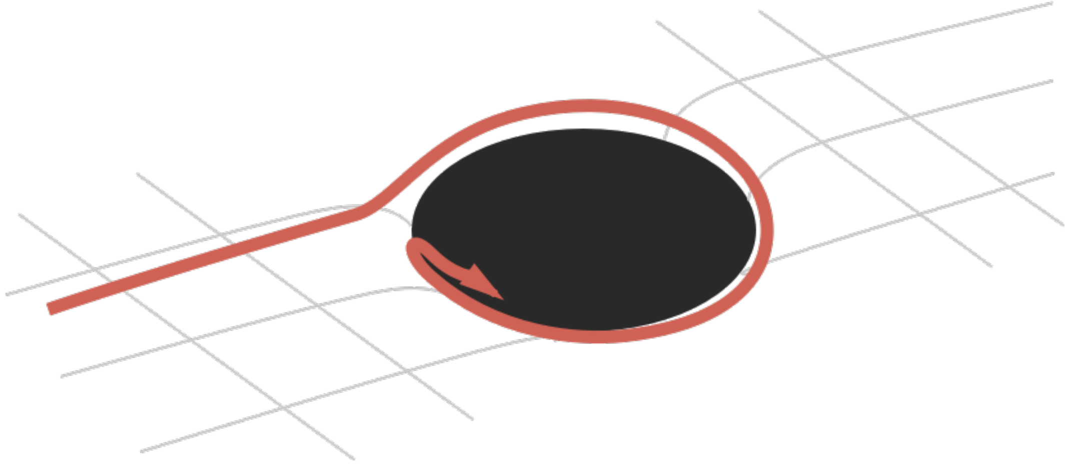

Figure 2.2. Light is falling inside the rotating black hole.

Now it is time to explain why this is happening. To understand this lets take a look at the Figure 2.3.

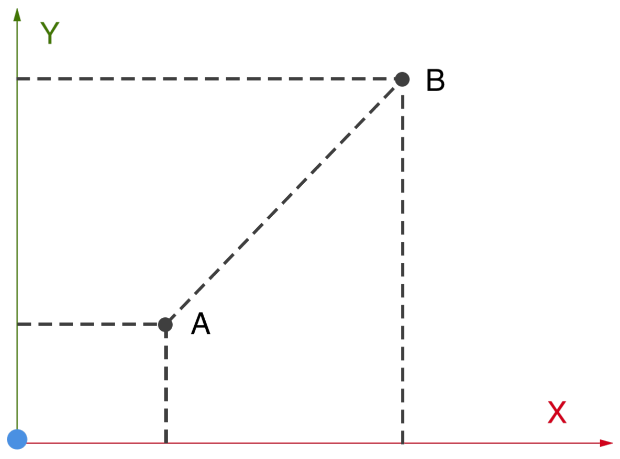

Figure 2.3. Distance between two points.

We have 2 points in the common Cartesian coordinate system. What we need to do is to measure the distance between those points. How we are going to do that? The answer is simple: we can calculate it using the Pythagorean theorem. And get the equation:

$$S^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$$

But what if we want to make a more general solution? We can define so-called *metric tensor*. For instance:

$$g_{i\,j} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

g is a simple matrix that describes Cartesian coordinate system in 2D.
Introduction of this metric allows us to define the previous equation like this:

$$S^2 = g\Delta x \Delta x$$

where

$$\Delta x = x^A - x^B$$

For better understanding lets change the geometry of the space from flat 2D to a curved surface of the Earth

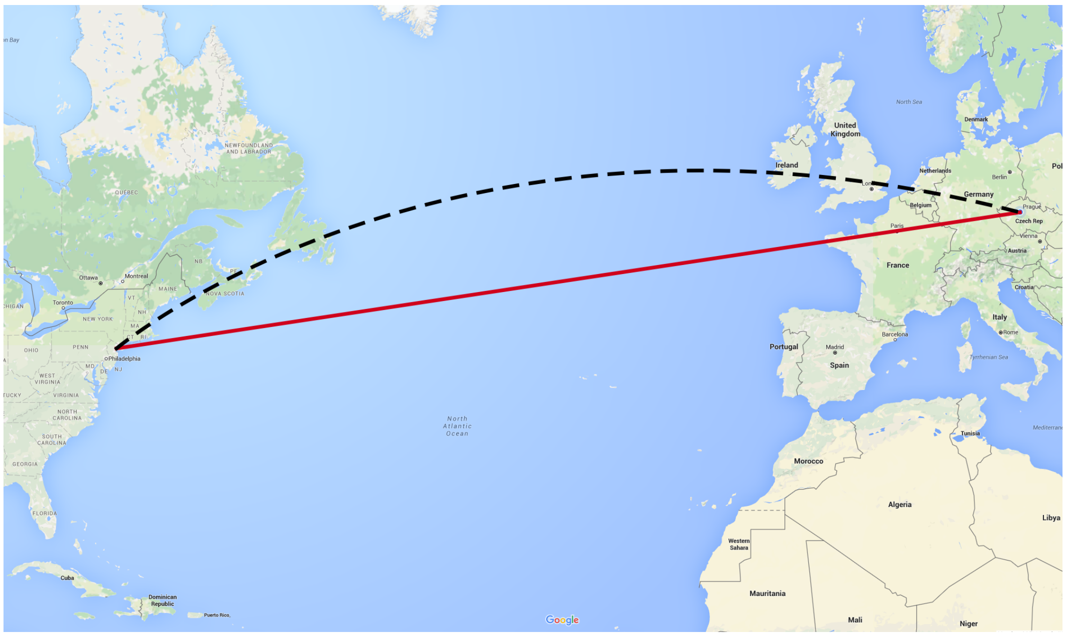

Figure 2.4. Distance between Prague and New York.

On the Figure 2.4 lets try to measure the distance between Prague and New York city. As we already know we can not directly pass a line between two points and measure its length as we are working now on the spherical or *geodesic* surface. However if we know the *metrics* we can easily fill the values that we have calculated for the Earth inside the matrix and calculate the distance.

$$g_{i\,j} = \begin{pmatrix} r^2 & 0 \\ 0 & r^2 \sin^2 \phi \end{pmatrix}$$

It will be approximately 6,568 km.

Now it is time to speak about Schwarzschild geometry. The Schwarzschild geometry describes the spacetime geometry of empty space surrounding any non-rotating spherical mass. One of the remarkable predictions of Schwarzschild's geometry was that if a mass M were compressed inside a critical radius r_s, nowadays called the *Schwarzschild radius*, then its gravity would become so strong that not even light could escape. The Schwarzschild radius r_s of a mass M is given by

$$r_s = \frac{2GM}{c^2}$$

where G is *Newton's gravitational constant*, and c is the speed of light [5].

With help of a Schwarzschild geometry we can define spacetime metric, that is called *Schwarzschild metric* :

$$g_{i\,j} = \begin{pmatrix} -1 + \frac{2M}{r^2} & 0 & 0 & 0 \\ 0 & (1 - \frac{2M}{r^2})^{-1} & 0 & 0 \\ 0 & 0 & r^2 \sin^2 \Theta & 0 \\ 0 & 0 & 0 & r^2 \sin^2 \Theta \sin^2 \phi \end{pmatrix}$$

This results that the relativity influence by the Earth is only about *3cm*.

So now we are ready to return to the black holes.

On the Figure 2.5 we can see the black hole with an accretion disk around it. Red filled part of the disk is the part that is situated in front of the black hole. The part filled with yellow is situated behind the black hole. White arrows represents the direction of the rotation of an accretion disk. As we can see the matter of an accretion disk goes to the top of the black hole, but in fact it doesn't. This is caused by the gravity lens effect that we were talking about.

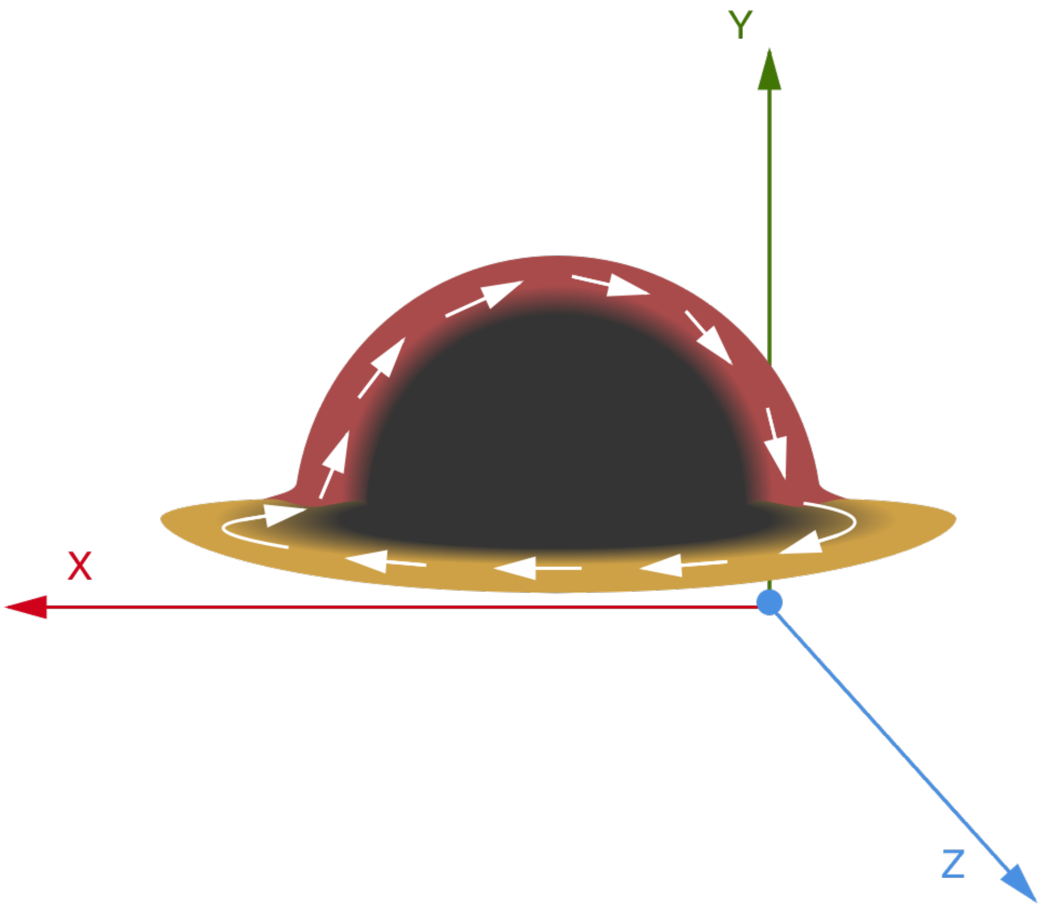

Figure 2.5. Accretion disk around the black hole.

If we will look at this black hole from another angle on the Figure 2.6, we will see that an accretion disk is still rotating strictly around the black hole. That happens because of a light distortion. In our application we are going to visualize this effect.

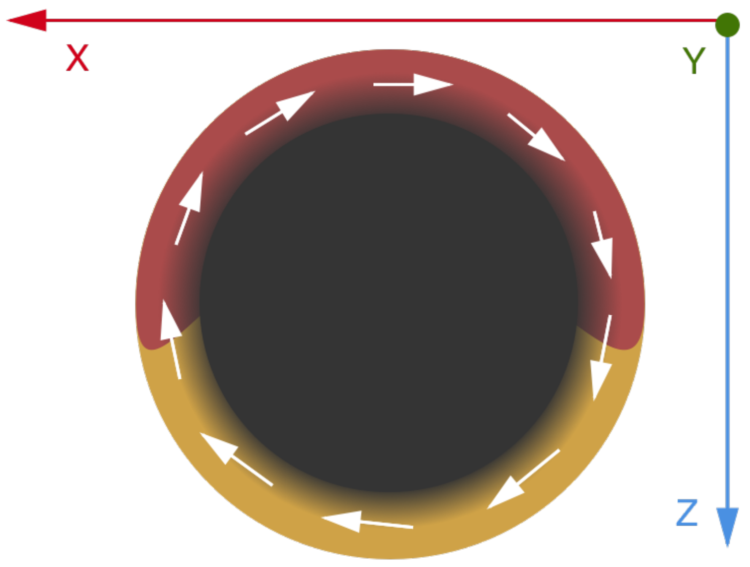

Figure 2.6. The accretion disk around the black hole viewed from the top

2.1.3 Scene structure

As we learned before if we look at the black from different angles we can see different views. But what if we move the camera around the black hole in XZ plane 2.7? The black hole will remain the same. Because it's accretion disk is symmetrical in X and Z axis. So the only important and interesting camera rotation for us is rotation in Y axis. However possibility to move camera around the horizontal plane still presents in the application.

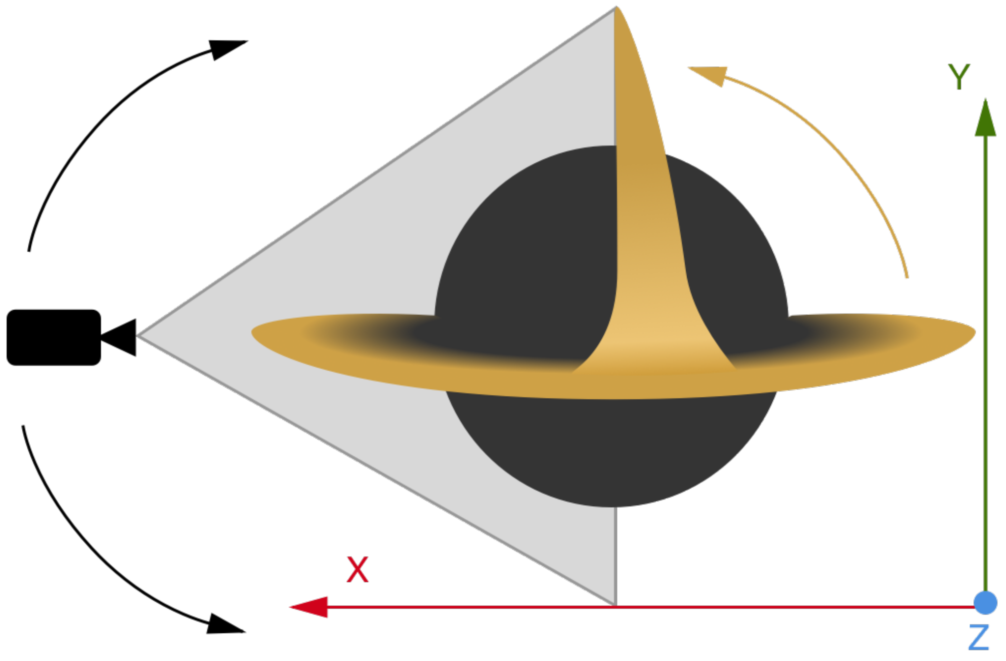

Figure 2.7. Camera position

Also we can move the camera around itself to watch the accretion disk from the side.

2.2 SIM5 library

The core of the application is the mathematical library [3] written by Michal Bursa. SIM5 library is an astronomical library written in C that provides such features as calculation of geodesics, ray-tracing, calculation of photon trajectories and interpolation. In fact this library provides the whole mathematical background of the black hole that we will use in our application. Lets take an overview of the functions and features that we use in our application.

2.2.1 Geodesic equations

A geodesic is a locally length-minimizing curve. Equivalently, it is a path that a particle which is not accelerating would follow. In the plane, the geodesics are straight lines. On the sphere, the geodesics are great circles (like the equator).[6]

To build such curve SIM5 lib offers geodesic structure, that we use to trace photons. To build this structure we need the following parameters: inclination angle, distance and coordinates of the observer. We will describe these equations with more details in the next section.

2.2.2 Ray-tracing

Raytracing requires fast and accurate computations of photon trajectories through the spacetime. There are plenty of possibilities how we can create a ray tracing. The choice between one or the other method depends mainly on the target application or the implementation requirements, also different methods provides different possibilities of what can be seen, when speaking about the black holes.

The first method is based on computation of the direction of the pseudo-angular-momentum. This is quite easy to implement and has relatively low computation complexity that makes this method very fast. However using this method we cannot create a real-time calculation of the accretion disk. We can only create the pseudo-Riemannian optics, this method is fully described in the Riccardo Antonelli's article [2].

Other solution is based on calculation of geodesic equations and integration for every pixel of the frame. This simplified geodesic is specified by the impact parameters of observer at infinity and the only parameter that we can change is the black hole spin relatively to the camera.

Next possibility is to create a geodesic ray-tracer that is specified by a point and direction (4-momentum vector) in the Kerr spacetime.[7] Kerr spacetime has two obvious symmetries that arise from the fact that its metric does not explicitly depend on time and azimuthal coordinate.[8] As it was already mentioned the Schwarzschild metric is applicable only on non-rotating black holes. So in order to create a ray tracer for the rotating black hole, we need to define another metric, which is the Kerr metric. In Boyer–Lindquist coordinates (t, r, θ, ϕ), the Kerr line element can be written as:[7]

$$ds^2 = -p^2 \frac{\Delta}{\Sigma^2} dt^2 + \frac{\Sigma^2}{\rho^2}(d\phi - \frac{2ar}{\Sigma^2} dt)^2 \sin^2\theta + \frac{\rho^2}{\Delta} dr^2 + \rho^2 d\Theta^2$$

with the definitions where we use units $G = c = M = 1$

$$\Delta = r^2 - 2r + a^2, \rho^2 = r^2 + a^2 \cos^2\Theta,$$

$$\Sigma^2 = (r^2 + a^2)^2 - a^2 \Delta \sin^2\Theta$$

$$a = \frac{J}{MC}$$

After the separation of the Hamilton–Jacobi equation [9] for geodesics we get

$$\int \frac{1}{\sqrt{R}} dr = \int \frac{1}{\sqrt{\Theta}} d\Theta$$

where

$$R = [(r^2 + a^2)E - aL_z]^2 - \Delta[Q + (L_z - aE)^2 + \delta_1 r^2],$$

$$\Theta = Q - [a^2(\delta_1 - E^2) + L_z^2 csc^2\Theta]cos^2\Theta$$

All this computations could be performed with help of SIM5 library, of course there are much more detailed computation inside of it, but for our purposes that knowledge is enough.

As we were talking about vectors of the observer let us define a photon *4-momentum vector* of an observer in his local frame that we will use for the ray tracing:

$$k = \begin{pmatrix} 1 \\ \sin\alpha\cos\beta \\ \sin x \sin\beta \\ \cos x \end{pmatrix}$$

We should notice that every calculation after this point are done in 4D spherical space. The first 3 components of this vector are classic x, y, z axis. But the fourth component is time.

$$(k_t, k_x, k_y, k_z)$$

Because of relativity objects around massive bodies could look different than we get used to. Because of a time delay we can see the image distortions. To be more exact, lets take a look at the Figure 2.8

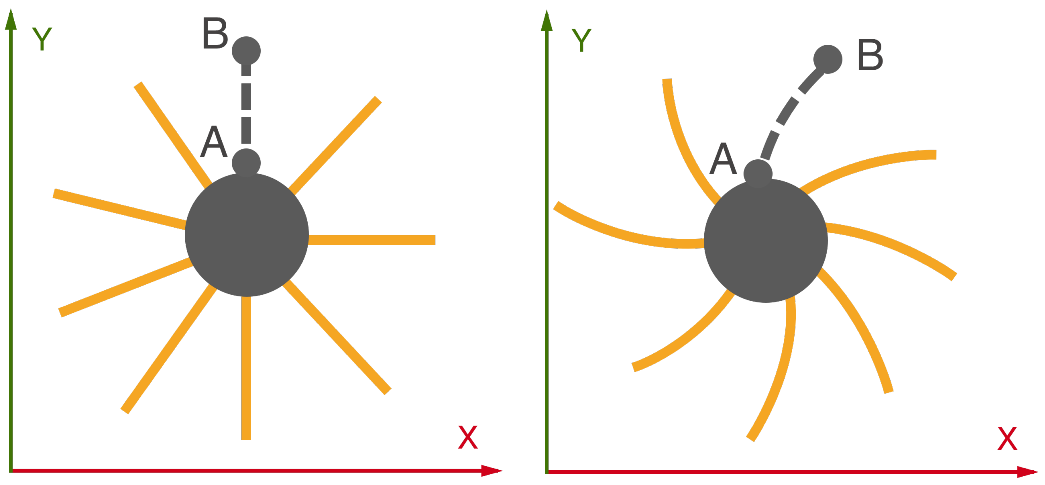

Figure 2.8. Visible distortions because of a time delay

As we can see at the left side of the picture is a space-time without time delay. We can see clear straight lines that comes from the center. But if we look at the right side we can see that it takes some time for a photon to reach its destination, so there is a visible light distortion there.

To make this ray-tracing we need to create a geodesic structure.

The geodesic structure contains information about the black hole's spin, observer's horizontal and vertical impacts, an inclination angle as well as cosine of inclination. Also it has geodesic parameters: Carter's constant, roots of R-integral, number of real roots of R integral, roots and coefficients of T-integral, radius of radial turning point (periastron) and the 4 momentum vector.[8]

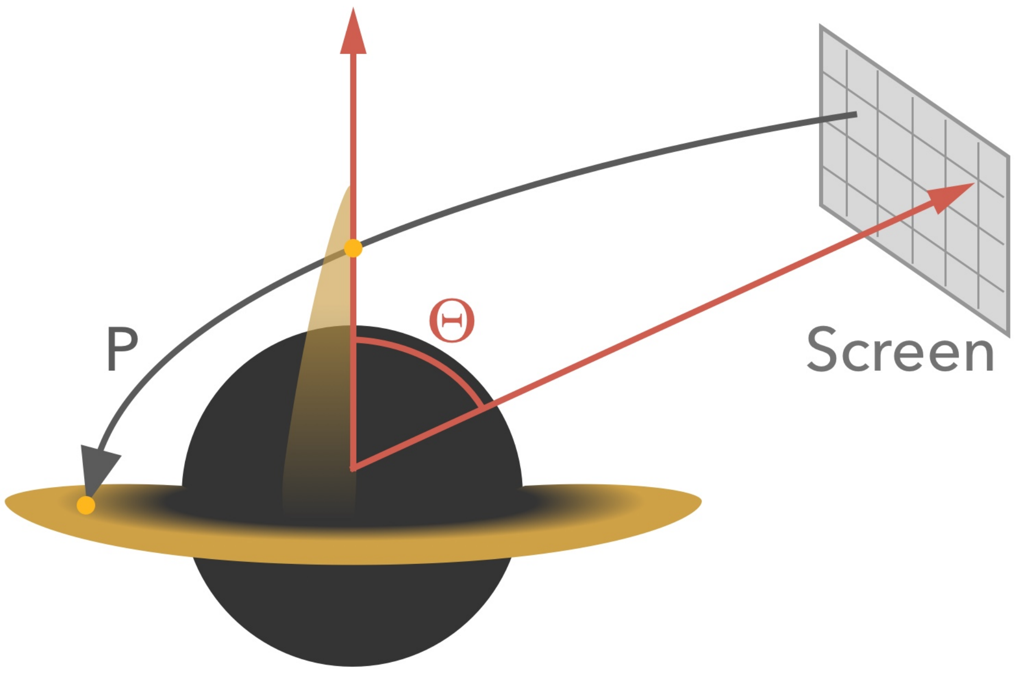

Figure 2.9. Finding a mid-plane crossing

After the creation of geodesic, we can start solving equation for every pixel. With help of *geodesic_find_midplane_crossing* function we can check whether the ray crosses an equatorial plane. The principles of this function are illustrated at the Figure 2.9, where we can see the screen clipping plane, the black hole model with an accretion disk around it and the vectors that are used to perform the ray tracing. Inside this function we determine orientation of momentum vector (poloidal component of it) and the following calculations are performed:

The left side of the equation that we have described at the beginning of this section

$$\int_0^\mu \frac{d\mu}{\sqrt{\Theta}} = P$$

And then the right side

$$P = \int_r^\infty \frac{1}{R} dr$$

Next we call *geodesic_position_rad* function with the position parameter that we've retrieved from *geodesic_find_midplane_crossing* function. Inside this function we calculate the radius at which the position integral gains value P. The return of this function is radius of disk intersection and after a simple comparison with the radius of last stable orbit and the inner disk we can decide whether light comes from that point or not.

After this step the ray tracing is done.

2.3 Choice of the motion controller

In this section we will speak about modern motion tracking technologies and devices. We will compare and choose the suitable hand tracking device that will fit our goals.

2.3.1 Microsoft Kinect

The first candidate is Microsoft Kinect. Kinect is Microsoft's motion sensor add-on for the Xbox 360 gaming console. The device provides a natural user interface (NUI) that allows users to interact intuitively and without any intermediary device, such as a controller. The Kinect system identifies individual players through face recognition and voice recognition. A depth camera, which "sees" in 3-D, creates a skeleton image of a player and a motion sensor detects their movements. Speech recognition software allows the system to understand spoken commands and gesture recognition enables the tracking of player movements. [10] Kinect comes with SDK for Windows and supports C++,C# and Visual Basic languages. Also there is an open source project called Open Kinect, that supports other platforms such as Mac OS and Linux.

2.3.2 Leap Motion

Our second candidate is Leap Motion controller. The Leap Motion system recognizes and tracks hands, fingers and finger-like tools. The device operates in an intimate proximity with high precision and tracking frame rate and reports discrete positions, gestures, and motion.

Figure 2.10. Leap Motion vs. Microsoft Kinect

The Leap Motion controller uses optical sensors and infrared light. The sensors are directed along the y-axis – upward when the controller is in its standard operating position – and have a field of view of about 150 degrees. The effective range of the Leap Motion Controller extends from approximately 25 to 600 millimeters above the device (1 inch to 2 feet).[11]

Leap Motion supports C++, C#, Java and many other languages. Also it supports almost every platform and most modern game engines.

2.3.3 Final choice

Microsoft Kinect is very good and precise system; however, it is probably too big and complicated for our goals. Also open source documentation is a bit overloaded. So in this circumstances Leap Motion seams to be a better choice. It is tiny, precise, has great structured documentation and fits our goals at 100 percent.

2.4 Analyses of the technologies

This section will unveil the technologies that we choose to develop our application. It will overview and explain a choice of the particular technologies such as CUDA, OpenGL and its frameworks.

2.4.1 Leap Motion API

The Leap Motion software runs as a service (on Windows) or daemon (on Mac and Linux). The software connects to the Leap Motion Controller device over the USB bus. Leap-enabled applications access the Leap Motion service to receive motion tracking data. The Leap Motion SDK provides two varieties of API for getting the Leap Motion data: a native interface and a WebSocket interface. [11] In our application we will receive the Leap Motion data using a native interface using a dynamically loaded library. So for final user the only thing that he or she have to do is to install Leap Motion drivers and connect the device while using the application.

The Leap Motion controller tracks hands, arms and tool that are visible in the camera's field of view. To handle theirs movement Leap Motion API has a Frame class that represents a frame that contains lists of tracked entities, such as hands, fingers, and tools, as well as recognized gestures. In fact the Frame object is the root of the Leap Motion data model.

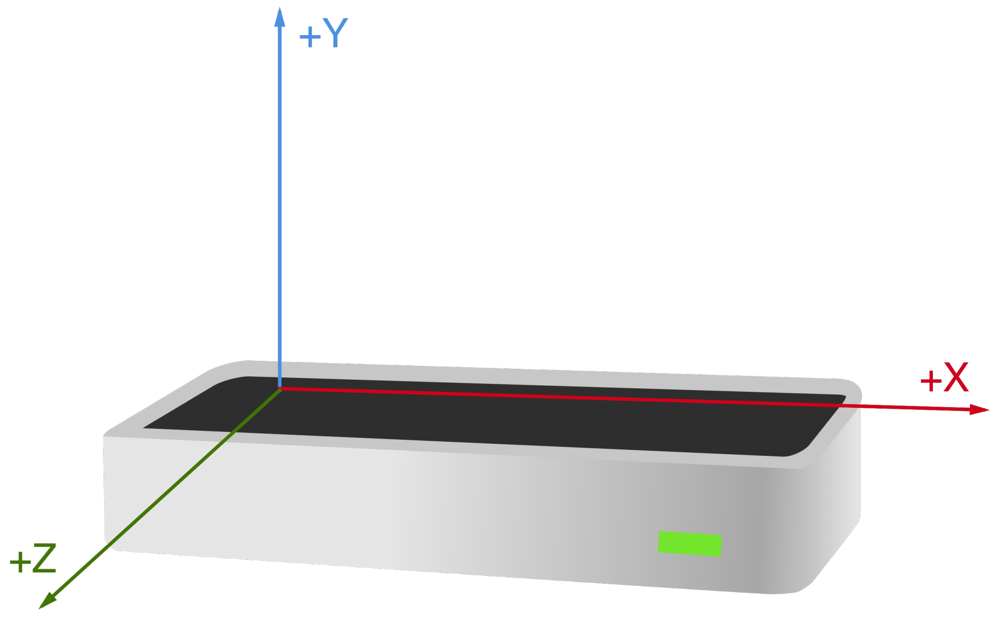

Figure 2.11. The Leap Motion right-handed coordinate system.

The Leap Motion system uses a right-handed Cartesian coordinate system. The origin is centered at the top of the Leap Motion Controller. The x- and z-axes lie in the horizontal plane, with the x-axis running parallel to the long edge of the device. The y-axis is vertical, with positive values increasing upwards (in contrast to the downward orientation of most computer graphics coordinate systems). The z-axis has positive values increasing toward the user. [11]

In our application we are going to use Hand model that provides information about position, orientation, speed, angle, etc. about user's hand. Hands are represented by the Hand class.

The next thing is a gesture detection. The Leap Motion API provides Gesture class that recognizes certain movement patterns of a hand. There are 4 basic types of gestures: [11]

- Circle — A finger tracing a circle
- Key Tap — A tapping movement by a finger as if tapping a keyboard key
- Screen Tap — A tapping movement by the finger as if tapping a vertical computer screen
- Swipe — A long, linear movement of a hand and its fingers
- Pinch — Detects the pinching between the thumb and any other finger
- Grab — Detects the grabbing strength of the user's hand

Each gesture contains its specific parameters such as MinLength, MinVelocity and more. This helps programmer to create precise gesture detection for more accurate usage in the application.

Also Leap Motion API provides *motions*. Motions are estimates of the basic types of movements inherent in the change of a user's hands over a period of time. Motions include scale, rotation, and translation (change in position). Motions are computed between two frames. We can get the motion factors for the scene as a whole from a Frame object. Also we can get factors associated with a single hand from a Hand object.[11]

2.4.2 NVIDIA CUDA

The performance of the application is very important. We have 2 possibilities how to perform a ray tracing: the first one is CPU based computation and the second using NVIDIA CUDA. Using the CPU provides us easy memory control and significantly simple application maintenance. Lets take a closer look at the CUDA technology.

CUDA is a parallel computing platform and programming model invented by NVIDIA. Every modern NVIDIA graphics card has this technology. Generally if we speak about parallelism the main goal of the programmer is to create a scalable application that will run on the multi-core processors. To do so usually we are using threads and processes. Two different threads could run in true parallel mode and could use physical cores of the CPU. However if we crate an application that will use a GPU for that purposes it is very complicated process. GPU provides us hundreds of graphic processors and to maintain this mechanism we need to continuously scale the parallelism of the application to get maximum performance of increasing number of processor of every particular computer. The CUDA parallel programming model is designed to help programmers in this challenge, allowing programmers with knowledges of C or Fortran programming languages to create applications with massive parallelism with very low entry-level of knowledges. There are three key abstractions - a hierarchy of thread groups, shared memories, and barrier synchronization - that are simply exposed to the programmer as a minimal set of language extensions. These abstractions allow threads to cooperate when performing a task and at the same time allow automatic scalability of the application.

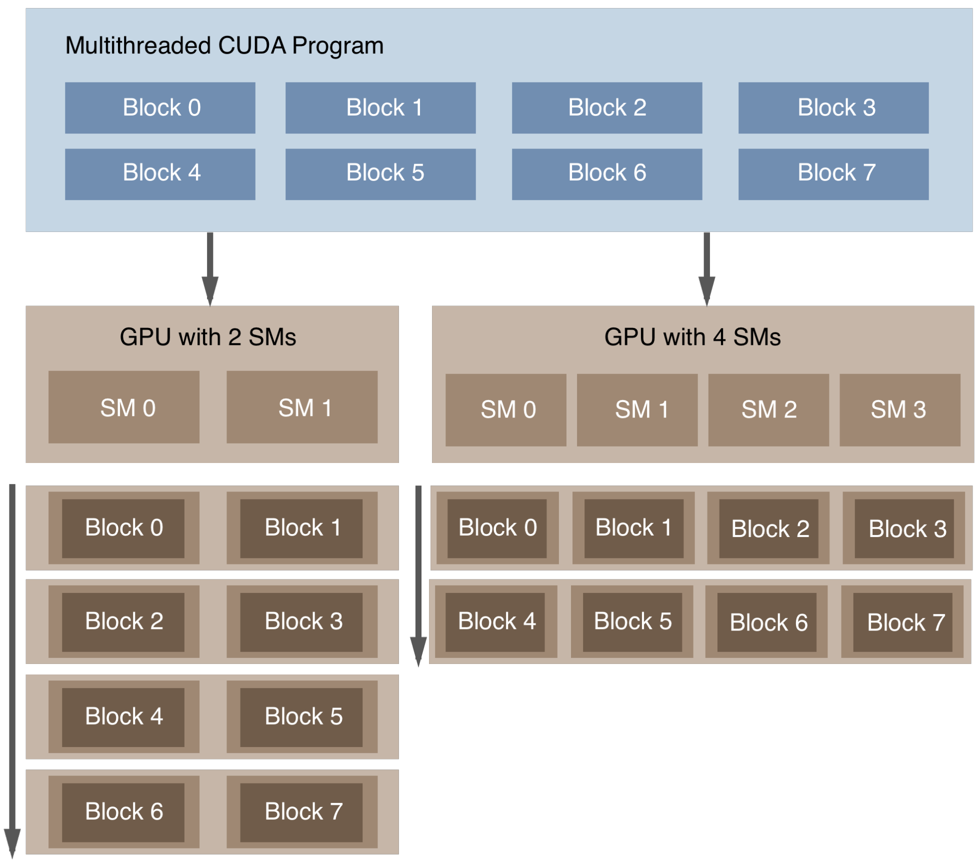

Figure 2.12. Automatic scalability

Indeed, each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of multiprocessors, and only the runtime system needs to know the physical multiprocessor count. The diagram 2.12 shows an abstract model of CUDA's scalability. A GPU is built around an array of Streaming Multiprocessors (SMs). A multi-threaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors [12].

As illustrated by Figure 2.13, the CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a co-processor to the host running the C program. This is the case, for example, when the kernels execute on a GPU and the rest of the C program executes on a CPU.

In our case to build a single frame of an application's output we need to allocate memory on the CPU side (host memory) and on the GPU side (device memory). Then we need to copy that memory to the graphics card and start the kernel. CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions.

Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime [12].

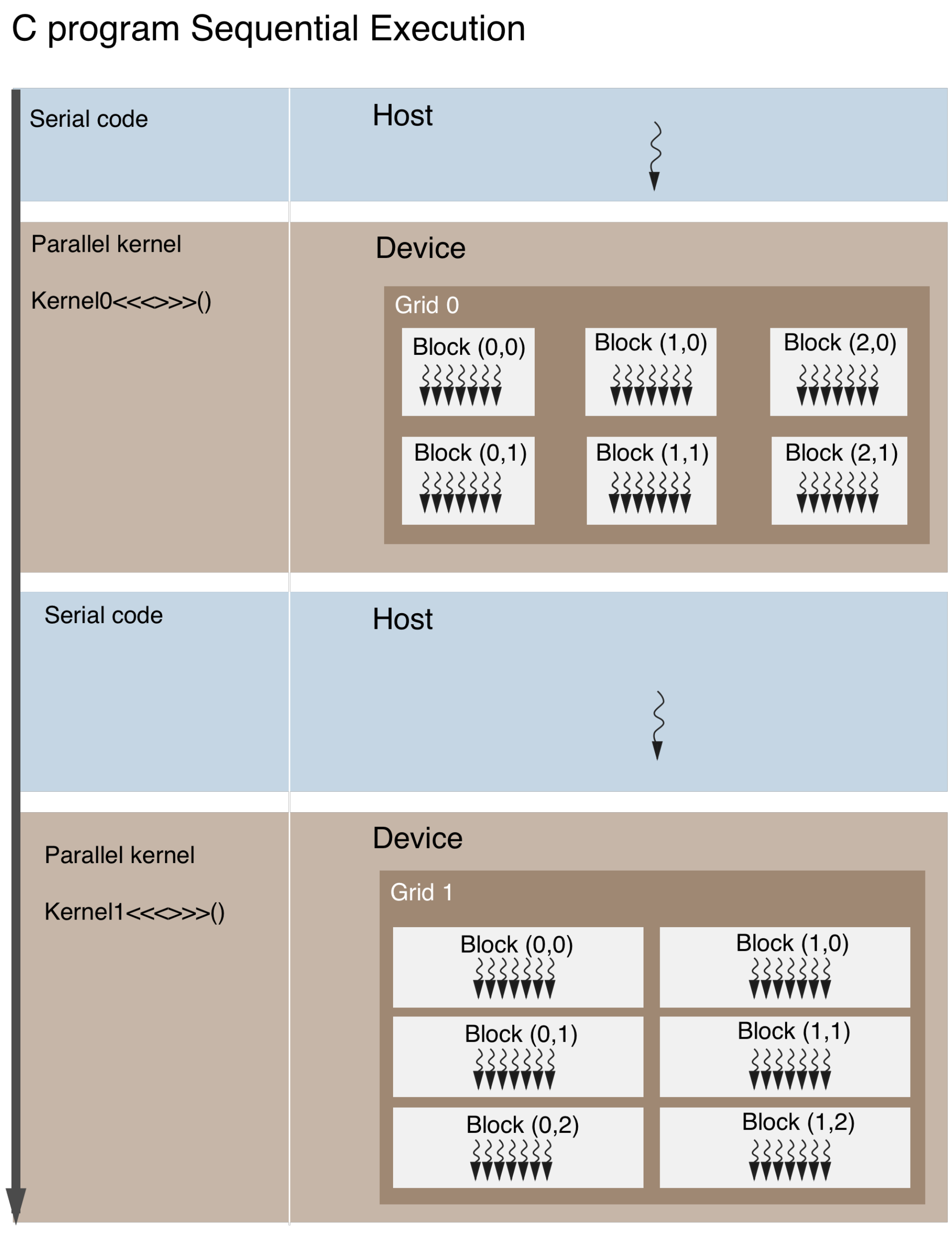

Figure 2.13. Heterogeneous Programming

Lets make a comparison of creation a frame using CPU and using CUDA.

```
for (iy = 0; iy < image_size; iy++) {
    for (ix = 0; ix < image_size; ix++) {
        double alpha, beta;
        alpha = (((double)(ix)+0.5)/(double)(image_dim)-0.5) * 2.0*rmax;
        beta  = (((double)(iy)+0.5)/(double)(image_dim)-0.5) * 2.0*rmax;
```

To create an image using CPU we need to loop over array of pixels and calculate an integral for each pixel.

But if we will try to perform this calculations using CUDA we need to copy the preallocated memory using cudaMallocPitch function and start the kernel. Where we get so-called pitches of an array and we can make computations on every row of that array.

```
      int ix = blockIdx.x*blockDim.x+threadIdx.x;
      int iy = blockIdx.y*blockDim.y+threadIdx.y;
      if ((ix>=N) || (iy>=N)) return;
      float alpha = (((float)(ix)+.5)/(float)(image_size)-0.5)*2.0*rmax;
      float beta  = (((float)(iy)+.5)/(float)(image_size)-0.5)*2.0*rmax;
```

We need to calculate the size of the blocks and the grid to reach the best performance. Unfortunately there is no precise way to calculate theirs size for every configuration, so we need to test multiple values that would be a compromise.

So as we can see CUDA seems to be the best solution for our purposes and probably the only one, if we want to create a real time ray tracing.

We will use CUDA Toolkit 7.5 which is the latest version at this moment.

2.4.3 OpenGL

OpenGL is a cross-platform API for rendering interactive 2D and 3D graphics applications. It supports many programming languages including C++, so it fits our choice quite well.

GLUT is the OpenGL Utility Toolkit, library that provides windows management features and monitoring of keyboard and mouse input. GLUT is a cross-platform library so our application will remain platform independent. With help of this library we will create a basic interaction with keyboard in our application [13].

Also OpenGL provides techniques and functions to work with shaders that are written in GLSL programming language and runs on GPU. We will use shaders for color adjustment and interpolation between frames. In our application we will use modern OpenGL 4.

To load the images that will be used in GUI textures we use Simple OpenGL Image Library (SOIL). That minimalistic library will help us load the .png or .jpg image files and store them to textures as an array of pixels.

Chapter 3
Application design

In this chapter we will speak about application's design, requirements and ways to complete them.

3.1 Requirements

The application will be used for educational purposes at Astronomical Institute of Czech Academy of Sciences. The main goal is to create an interactive and highly entertaining application for adults and kids that will show them what is a black hole and how does it look from different angles. From this goal we can derive functional and non-functional requirements.

Functional:

- black hole rendering using SIM5 library
- change observer's position using keyboard
- create real-time light-movement animations around the accretion disk
- turn on/off effects (photons, background, color) in real time
- use NVIDIA CUDA for calculations
- use Leap Motion to control the camera

Non-functional:

- application should be entertaining
- easy-to-use
- acceptable frame-rate at least 25 fps
- minimal resolution at 720p
- fast and simple application execution

3.2 Programming language and platform

To keep code clean and uniform we choose C++ programming language. This allows us to use SIM5 library without any conversions and the code of our application remains fast and extendable. Also C++ is a very flexible programming language, that allows to control memory and threads, which can help us to boost up a computing time of the black hole mathematical model. The programming language C++ has great compatibility with OpenGL and NVIDIA CUDA technologies as well. So for our purposes C++ is an ideal choice.

As a main platform we choose Ubuntu Linux, because of its freeness and GNU license profits. In fact our application remains cross-platform, and it can be compiled on any operating system without complications.

3.3 Structure overview

3.3.1 Sequence diagram

The key of the structure of the CUDA part of the application is minimalism. All the GPU kernels and host functions are stored in a single file. This allows easier maintenance and extendability of the application.

This allows us to easily precompile the CUDA part of the program and declare only necessary functions, that we will use in C++ part in a single header file. Also in this part we connect SIM5 library, with help of *sim5-lib.h* file. That contains all the necessary internal includes of the library.

In C++ the basic structure of the application is represented by classes. This provides us clean and well structured code, that we can easily extend and modify. To connect the CUDA part of the application we simply include a single header file. After this we are ready to make computations and can send parameters to the CUDA's side.

The Figure 3.1 describes the process of a single frame creation and shows how the different classes interacts with each other during this process. More detailed description of each class follows next in this chapter.

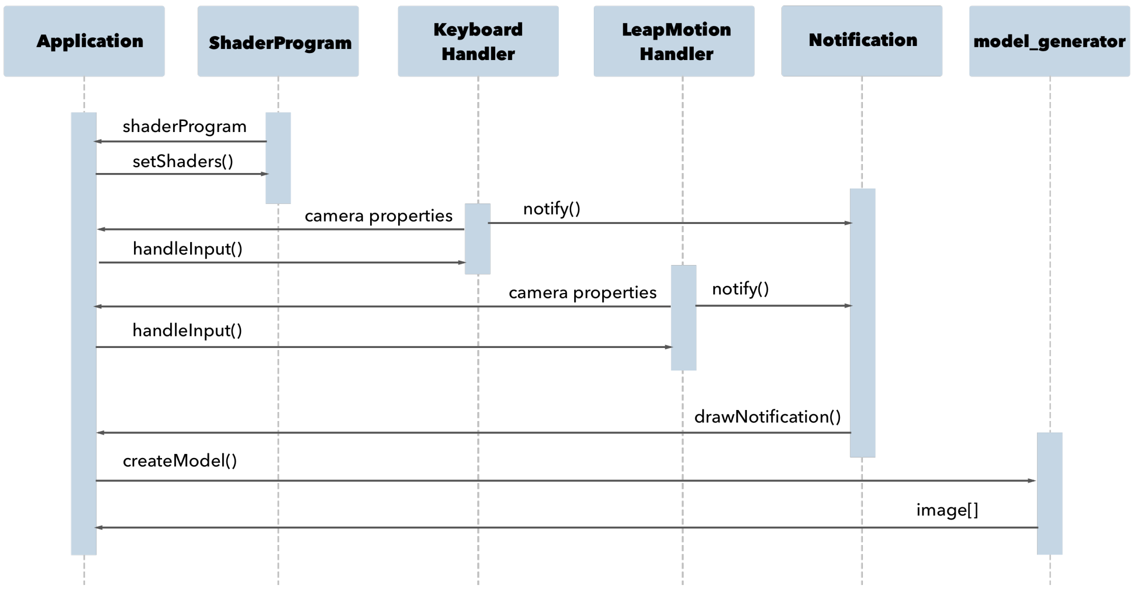

Figure 3.1. UML Sequence diagram

3.3.2 Compilation of multiple languages

In the application we will use multiple languages and compilers. NVIDIA CUDA's compiler *nvcc* compiles sources with .cu extension so we must create a simple Makefile that will precompile .cu source file that contains CUDA part of the application and will output it to the .o object file that will be used in the C++ part of the application. After this we need to link the created .o file to use it in g++ compilation. G++ is a C++ compiler that we use in our application. Final stage of our Makefile is to remove all the unnecessary files left by compilers. After this steps we will receive a single binary file of our application that we can easily execute.

3.4 Leap Motion gestures

To achieve our goal we need to add gesture control to the application. Gestures should be intuitive and easy to produce. Basically we have the camera rotation, zoom-in/zoom-out and turn on/off the accretion disk movements. The reason why we have chosen these gestures is described in the Chapter 5.

The first one is so-called circular gesture that turns off or on the spots rotation around the black hole. This is gesture is illustrated on the Figure 3.2. When user rotate his finger around its axis the light spots are turning off or on.

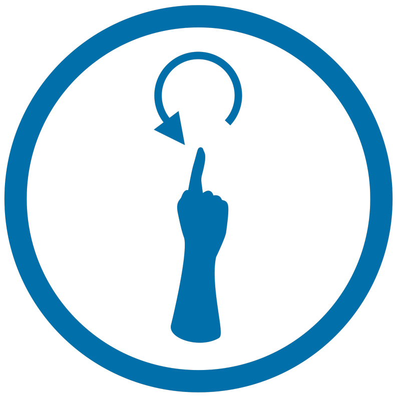

Figure 3.2. Circle gesture

The next type of gesture is a scale gesture that is illustrated on the Figure 3.2. When user produces grab gesture with both hands and move them from the center to the sides from each other, the camera moves forward producing camera's zoom in action. If the user move his hands like if he wanted to put them together and keeps grabbing his both hands camera zooms out. Those gestures are analogical to the pinch-to-zoom gestures in 2D that could be produced with help of fingers, but represented in 3D space with hands.

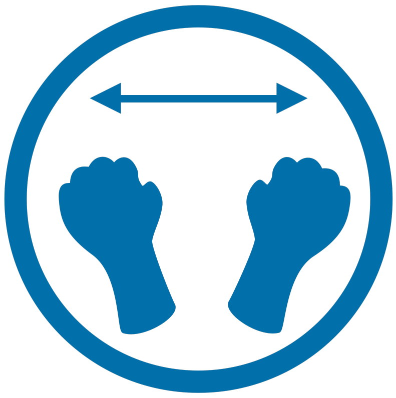

Figure 3.3. Scale gestures

The third gesture is a camera rotation gesture that is illustrated on the Figure 3.4. In order to initiate the rotation of the camera around its own axis user should move his hand and then hand translation reflects the change in rotation of the camera so user can look around the black hole in different axis.

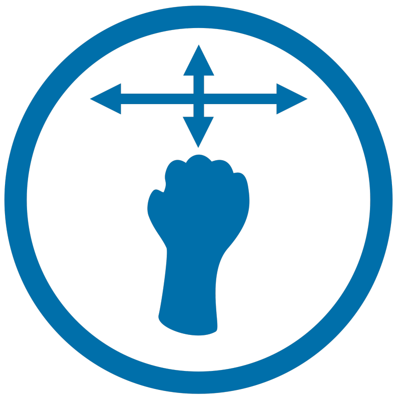

Figure 3.4. Camera rotation gesture

And the last gesture is the inclination changing gesture that is illustrated on the Figure 3.5. To activate this gesture user must pinch his thumb finger with any other finger and move his hand in Y axis. If user moves his hand up or down the camera rotates around the black hole that makes possible to see it from the top or bottom.

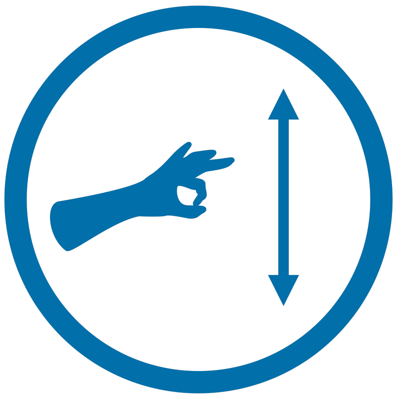

Figure 3.5. Inclination gesture

All of those gestures were selected after the testing with real users, where were 3 different variations of the gestures. These gestures seemed for testers very intuitive and easy-to-use. Also from the programmers point of view those gestures are not too complicated to implement so this solution fits our goals. More detailed overview of the testing is described in the Chapter 5.

Also every icon that was shown in this section is used in the the application GUI, to make a sensible feedback when user starts interaction with the application. These icons makes the application self-describing and even more intuitive.

Chapter 4
Implementation

This chapter presents implementation of the application. Here will be detail explanations of the most important parts of code as well as a general overview of the whole application runtime.

4.1 Application class

The application class is the starting point of the program. In this class the application instantiate the motion listener class, shader loader and links header file that contains CUDA functions. Also here are situated the main loop and keyboard listener of the GLUT window manager. In the application class we instantiate the shader program class and setup the textures as well. After setting up window's default width and height we initialize black hole default parameters that the program will use after the start of the application. Next thing is the texture setup, inside which we are going to draw our black hole model.

4.2 Camera structure

The Camera structure provides the position, inclination, width of view, and light spots flag. All these fields describe the future image of a black hole and are required when generating the model. Light spots flag was placed here to minimize the input parameter of the *calculatedImage()* function and easier handling in *LeapMotionHandler* and *KeyboardHandler* classes. You may ask why do we use a structure instead of class when we are working with C++. The answer is simple, this structure is used also in the CUDA part of the application, that receives it as parameter. So we don't have to send the raw variables between two languages. C++ structures are fully supported by the C compilers and if we don't add C++ only features such as member functions and classes the structure will remain multilingual.

4.3 Notification class

The Notification class provides simple minimalistic notifications. Contains such methods as *drawNotification()* and *setNotification()* that are called from the input handling classes, when user interacts with the application. Notification is a simple icon that describes currently selected gesture and what will happen when user will change its hand position. All those icons were described in the Section 3.4. When the handler class detects the user input it calls the notification method that sets initial time of the notification and draw the notification icon at the top left corner of the application window for the specified time (that time is a constant and stored in the *Constants.h* file).

4.4 Leap Motion Listener and Handler

The leap motion handler class contains plenty of useful methods, such as onConnect, onFrame, onExir, and so on, that maintain gestures and hand movements independently from the main loop of the rendering part of the application by using a different thread. Lets make an overview of the most important methods of this listener. The first one is *SampleListener::onConnect*, this method is called when the leap motion controller has been connected and enables gesture detection and specifies which gestures will be used. For instance: *controller.enableGesture(Gesture::TYPE_SWIPE)* enables detection of a *swipe* gesture.

The next method is *SampleListener::onFrame*, but for our purposes we are not going to use it as listener. As it was already mentioned the Listener runs as a single thread and all other application variables are located in different classes. This causes quite complicated cross-thread handling for us. We need to synchronize all those threads and manipulate with variables very carefully. To avoid this we will implement *onFrame* method inside the LeapMotionHandler class, where we handle the frame rendering, because we don't need to calculate gestures or track hands before we can actually see the result. As we already know Leap Motion API is using a Frame model, so this method gets the most recent frame from the controller instance, which will be global in this case and report some basic information. Also for further *Motions* handling we need to store the previous frame, or in our case the 3rd frame before current. The Frame instance allows us to do that simply inserting the parameter inside the *controller.frame()* method. We have chosen the 3rd frame for better recognition and frame drop prevention. However this value could be up to the maximum age 59.

```
const Frame frame = controller.frame();
const Frame previousFrame = controller.frame(3);
```

After this we need to get the HandsList object if the hand presents in the current frame.

```
HandList hands = frame.hands();
```

From this point we can get the basic information about hand's position, direction and get access to the hand's finger objects. To handle the inputed gesture or manipulation with hands first of all we need to detect how many hands are in the frame, to do so we call *hands.count()* method from the HandList object. If the number of hands is more than 1 that means that we should handle the scale gesture, as it is the only gesture that uses 2 hands. But if we will start to scale the picture from the first detection of 2 hands this may cause a usability problem. Because the camera will start to zoom in and zoom out even when user doesn't want to scale the image, for example if user had zoomed in and now he wants to rotate the camera, so he need to move the second hand away. But we still have 2 hands in the frame for that moment and further calculation will produce an unwanted scaling. To prevent this we need to check if user had grabbed both his hands, and only after this we will start the scaling. To do so we call *float grab = hand.grabStrength();* method and store it in a local variable. This parameter is the holding strength of a hand grab. The strength is zero for an open hand, and blends to 1.0 when a grabbing hand pose is recognized. In our case we use *0.9* grab strength to prevent an unwanted grab gestures. After the successful grab detection we get the actual direction of both hands that could be done using the *hand.translation(previousFrame)* method.

```
Leap::Vector linearMovementLeft = leftHand.translation(previousFrame);
Leap::Vector linearMovementRight = rightHand.translation(previousFrame);
```

The return value of this method is a Leap::Vector. This vector represents the change of position of the hand between the current frame and the specified previous frame. The *Leap::Vector* structure represents a three-component mathematical vector or point such as a direction or position in three-dimensional space.[11] After this we need to get the magnitude or length of this vector to get the scale factor (how much the image will be scaled). This could be done with help of *linearMovement.magnitude()* method. After we received the needed values we need to multiply the linear movement vector's X value of the left hand by the length of that vector and then sum it with the linear movement vector's X value of the right hand and its length. The calculated value represents the current zoom in millimeters so in order to use it in our application's metrics we need to divide it by 20. This value was calculated after the tests. After this we are ready to store new zoom in our camera's zoom field and wait for the next input.

```
camera->zoom+=(linearMovementLeft.x*linearMovementLeft.magnitute())
+ (linearMovementRight.x*linearMovementRight.magnitute());
```

Also we don't want to simultaneously detect the scale and camera rotation motions so when we detect 2 hands in the frame we deal only with scaling skipping other motions.

The same process is required to compute the camera rotation gesture; however, as we decided to use a hand grab to activate the scale mode we also will use the grab gesture to activate the camera rotation action for the same reasons. For this we check the previously calculated value of *grabStrenth* and if it is greater than 0.9, that means that we should complete the camera rotation. The only difference is that we need to compute the linear movement vector of one hand.

```
if(grabStrength > 0.9){
  camera->y += ((linearMovement.magnitude()*linearMovement.y)/1000);
  camera->x += ((linearMovement.magnitude()*linearMovement.x)/1000);
}
```

To calculate whether there were any inclination changing movements or not we need to detect if there was a pinch gesture in the scene. To do so we call the *hand.pinchStrength();* method and if it is greater than 0.9 that means that the pinch gesture was produced and we simply multiply the translation vector's magnitude by the translation vector' Y value and divide this result by 1000 to fit our application's metrics. The magnitude of the translation vector is the L2 norm, or Euclidean distance between the origin and the point represented by the (x, y, z) components of this Vector object.[11]

```
camera->inclination+=(linearMovement.magnitude()*linearMovement.y)/1000;
```

After this we need to check if there was a Circular gesture in the frame. To do so we create a GestureList object and with help of simple *switch* check the type of the gesture.

```
const GestureList gestures = frame.gestures();
```

After the gesture was detected we check the direction of finger rotation in case of the circular gesture and set the boolean variable, that represents whether the light spots animation is enabled or not.

```
if (circle.pointable().direction().angleTo(circle.normal()) <= PI / 2) {
    camera->lightSpots = true;
```

4.5 KeyboardHandler class

We use the Leap Motion controller to interact with the application; however, we still can use the keyboard to do so. The KeyboardHandler class handles user's input from the keyboard and changes the scene parameters as well as LeapMotionHandler. All the keys are described in the table below 4.1.

Key	Action
W	Increase inclination angle
S	Decrease inclination angle
Q	Zoom in
A	Zoom out
H	Rotate camera up
G	Rotate camera down
X	Rotate camera left
C	Rotate camera right
Z	Reset the scene
F	Turn off/on light spots

Table 4.1. Keyboard keys for interaction.

KeyboardHandler uses simple GLUT key listener function and with help of a camera structure and switch over the pressed key changes the scene parameters.

4.6 ShaderProgram class

The Shader Program class contains a vertex shader object, fragment shader object and the compiled shader program. After instantiation of the Shader Program class the constructor is called, which receives as argument the type of the shader and the source of the current shader as string. In the constructor the shader program is compiled, attached and linked for further using in the application.

4.7 Model computation

The computation of the black hole model takes it place inside the CUDA's part of the application.

4.7.1 Creating a frame

First of all we need to allocate memory for an array of pixels that will be processed in future by the GPU side of the application. The size of the array is derived from the C++ part of the application and equals size of the application window.

```
float *image = (float*) calloc(image_size * image_size, sizeof(float));
```

Then we need to start so-called spot routine, to create a light spots around the black hole. Those spots will rotate until the turn-off gesture will be detected. After that we need to set up the corresponding inputed parameters such as camera direction and position and set them at the 4-momentum vector that will be used for ray tracing. Next we need to prepare the GPU memory and copy an empty array of the future pixels. Also after each action we need to handle the errors that may occur. Very important step here is that we need to allocate memory only once, because the memory allocation for every frame takes significantly more time than allocation at the beginning of the execution.

```
size_t pitch;
cudaMallocPitch((void**)&d_image, &pitch, N*sizeof(float), N);
err = cudaGetLastError();
if(err!=cudaSuccess) fprintf(stderr);
image_output[0]=1.0;
cudaMemcpy2D(d_image, pitch, image_output, N*sizeof(float),
N*sizeof(float), N, cudaMemcpyHostToDevice);
err = cudaGetLastError();
if(err!=cudaSuccess) fprintf(stderr);
```

Finally after this setup we calculate the block and grid sizes and start the kernel.

```
dim3 grid(iDivUp(N,BLOCKSIZE),iDivUp(N,BLOCKSIZE));
dim3 block(BLOCKSIZE,BLOCKSIZE);
gpu_kernel<<<grid,block>>>(a, inc, rms, rmax, d_image, pitch, N, t,
image_size, spots);
```

To calculate the grid and block sizes we use a simple equation *((a % b) != 0) ? (a / b + 1) : (a / b)*, where a is the initial size of the block and b is the initial size of the grid. For our purposes we use the same size of the block and grid and it equals 16. This number was chosen after testing the performance and it showed the best results.

Inside the kernel we are executing the ray tracing which theory was described in the 2.2.2 section.

4.7.2 Ray tracing

Ray tracing starts inside the kernel.

```
__global__ void gpu_kernel(float a, float inc, float rms, float rmax,
float* d_image, size_t pitch, int N) {
    int ix = blockIdx.x*blockDim.x+threadIdx.x;
    int iy = blockIdx.y*blockDim.y+threadIdx.y;
    if ((ix>=N) || (iy>=N)) return;
```

ix and iy variables are indices of the inputed array of pixels. To get them we need to multiply the block id and the block dimension and after this calculate the sum of the result and the thread id.

After calculation of the indices we need to compute the impact parameters

```
float alpha = (((float)(ix)+.5)/(float)(image_size)-0.5)*2.0*rmax;
float beta  = (((float)(iy)+.5)/(float)(image_size)-0.5)*2.0*rmax;
```

these parameters are just image coordinates scaled to rmax and shifted so, that the origin of the black is in the center of the image. Next we declare a 4-momentum vector and create the rotation matrices. After the creation we multiply particular rotation matrices from the right by 4-momentum vector to adjust the rotation of the camera.

```
n[0] = (-alpha / sqrt(sqr(r_obs) + sqr(beta) + sqr(alpha)));
n[1] = (-beta / sqrt(sqr(r_obs) + sqr(beta) + sqr(alpha)));
n[2] = (-r_obs / sqrt(sqr(r_obs) + sqr(beta) + sqr(alpha)));
double rotateX[3][3], rotateY[3][3];
```

Also we need to transform this vector to coordinate frame with help of *on2bl(n2, k, &tetrad_obs);* function. After we create *geodesic gd* instance, where the calculated integrals will be stored. Next we initialize photon trajectory for particular impact parameters.

```
geodesic_init_src(a, r_obs, m_obs, k, 0, &gd, &status);
```

After this we call *geodesic_find_midplane_crossing(&gd, 0);* function to find where the photon crosses equatorial plane and next we get radius of disk intersection from the position parameter

```
r = geodesic_position_rad(&gd, P);
```

After this we iterate over the spots array. This part has the highest complexity in the whole frame creation process. Because we must iterate over the array of the spots in every frame for each single pixel and change the pixel brightness according to the spot brightness. Also we should handle each spot after each frame of the application when it is time to remove the old spot and create a new one. This process is implemented in the host side of the program. However, after the calculation the brightness of the spots we need to compare the radius of the black hole and radius of a geodesic in order to detect where the accretion disk cannot exist, thus radiation is zero. Also we should to apply the relativistic correction to the disk and calculate the brightness of the photons that make one orbit around the black hole and may bring light from the bottom side of the disk. After this we can finally calculate the result and store it to the pixels array.

4.8 Rendering

The rendering part of the application is located in the Application class.

After we retrieve an array of float values that indicates brightness of each pixel of a black hole we write those values to the texture with help of *glTexImage2D()* OpenGL function and map the texture through the whole application window. After that we free up memory to prevent its leaks. These steps repeat for every single frame.

```
    glEnable( GL_TEXTURE_2D );
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, image_size, image_size,
    0, GL_LUMINANCE, GL_FLOAT, image);
    free(image);
```

Also here we are starting to use our shader program, that was compiled earlier in the Shader Program class. Using the OpenGL function *glUniform1i()* we bind our texture to be processed with the shader program.

```
glUniform1i(glGetUniformLocation(shaderProgram, ''texture1''), 0);
```

Inside the vertex shader we simply assign and pass the texture and variables to the fragment shader.

```
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 color;
layout (location = 2) in vec2 texCoord;
```

In fragment shader we finally compute the output color of each fragment.

```
color = texture(texture1, TexCoord)* vec4(modColor, 1.0f);
```

Chapter 5
Testing

In this chapter we will see the final result of the application's implementation and how CUDA can improve the performance. Also we will describe the usability test and overview the usability problems that were found.

5.1 Gestures tests

Gestures are the main input method of the application. So in order to provide the best experience for users the first test was a gesture test. Gesture tests was similar to the usability test, but they were performed by the developers. The goal of this test is to detect usability problems at the application design stage. These tests took place when the application was at beta version. 2 participants was selected to decide which gestures will be used in the application. Those participants had a basic knowledges about the application and what it should do, also they had experience with interactions with Leap Motion controller. First of all the table of gestures and actions was created and after this started the implementation of all of those gestures. The following Table 5.1 shows which gestures were selected to be tested.

#	Gesture	Action
1	Swipe up	Increase inclination angle
2	Swipe down	Decrease inclination angle
3	Grab and move and zoom	Zoom-in/out and camera rotation
4	Pinch and move	Inclination and camera rotation
5	Grab and zoom with 2 hands	Zoom-in/out
6	Grab and move	Rotate the camera
7	Pinch and move in Y axis	Change inclination angle
8	Circular gesture	Turn off/on light spots
9	Circular gesture	Zoom-in/out
10	Detect perpendicular palm and move	Camera rotation

Table 5.1. Gestures for interaction.

After the testing 4 gestures were selected for further processing: **#5, #6, #7, #8**. They were used in the final version of the application. The other gestures had the following problems:

- Gestures **#1** and **#2** that were using the provided Swipe Gesture of the Leap Motion API showed that swipe constant motion is too unpredictable and not precise. In order to activate them user had to move his hand fast and we can only change the inclination linearly.
- Gesture **#3** showed that the zooming with help of pull and push actions is not the most precise and easy gesture, when user tries to zoom in an unwanted camera rotation happens that makes the usage of the zoom function too complicated.

- Gesture **#4** was not precise when the user moved his hand to the sides. Leap Motion controller detects pinch gesture with thumb and any other finger, so when the hand moves to the sides, sometimes an unwanted camera rotation happens. However pinch and move in Y axis gesture was fine and precise so it was selected for further processing.
- Gesture **#9** seemed not to be intuitive and precise.
- Gesture **#10** is not the most comfortable gesture to produce, also it would be difficult to implement. Because we had to calculate the palm's normal to detect whether there were any motion gestures.

Pinch and grab gestures are so-called activators, after theirs detection we start the motion processing. We are using them for a reason. The reason is that we don't want to detect user's hand motions every time the hand appears in the frame. It would be not comfortable to sit near the table with constantly detecting controller and every motion with a hand will evaluate the camera movement. So this could make the interaction with the application almost impossible.

5.2 Application output overview

After the application start the camera is placed in front of the black hole. The light spots are activated and user can rotate the camera, zoom in or zoom out as well as change the inclination angle.

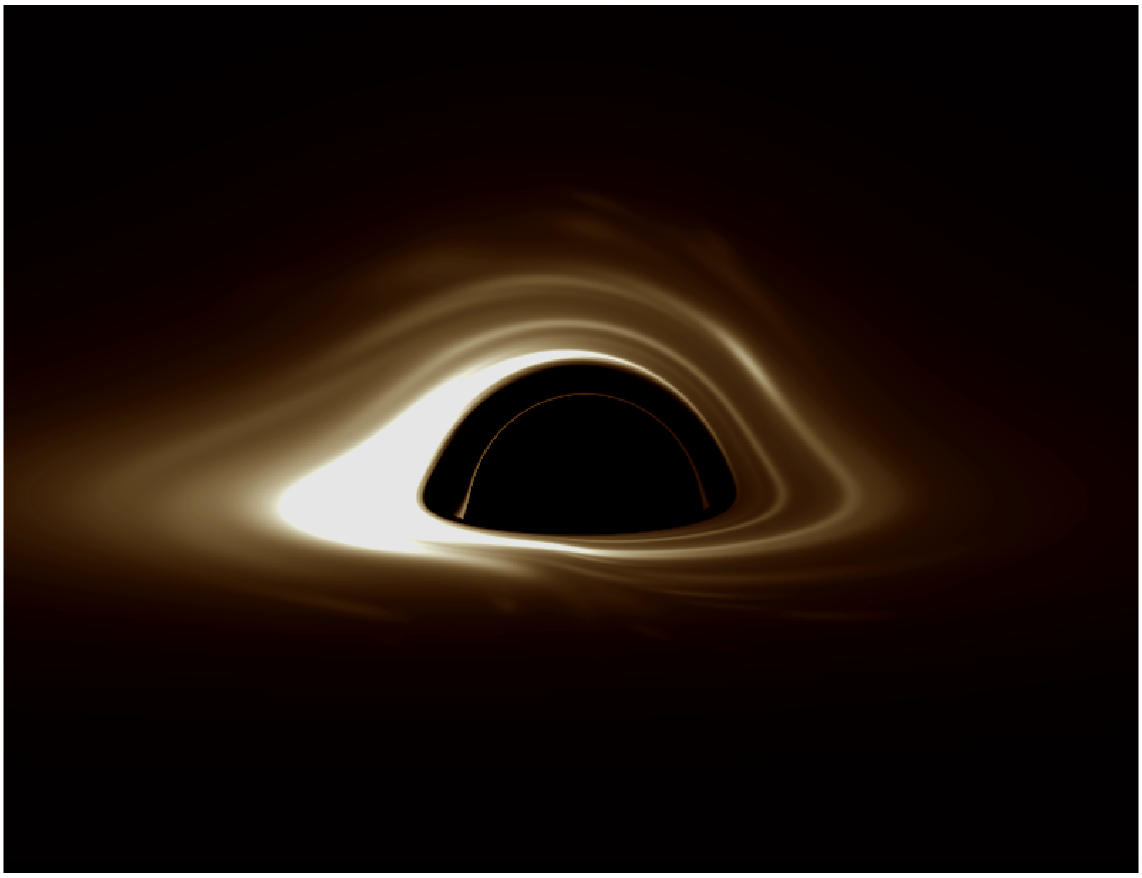

Figure 5.1. Initial position of camera and black hole's accretion disk.

If we'll change an inclination angle we can see the black hole from the top. Next picture 5.2 shows how the photons of light are continuously falling inside the black hole.

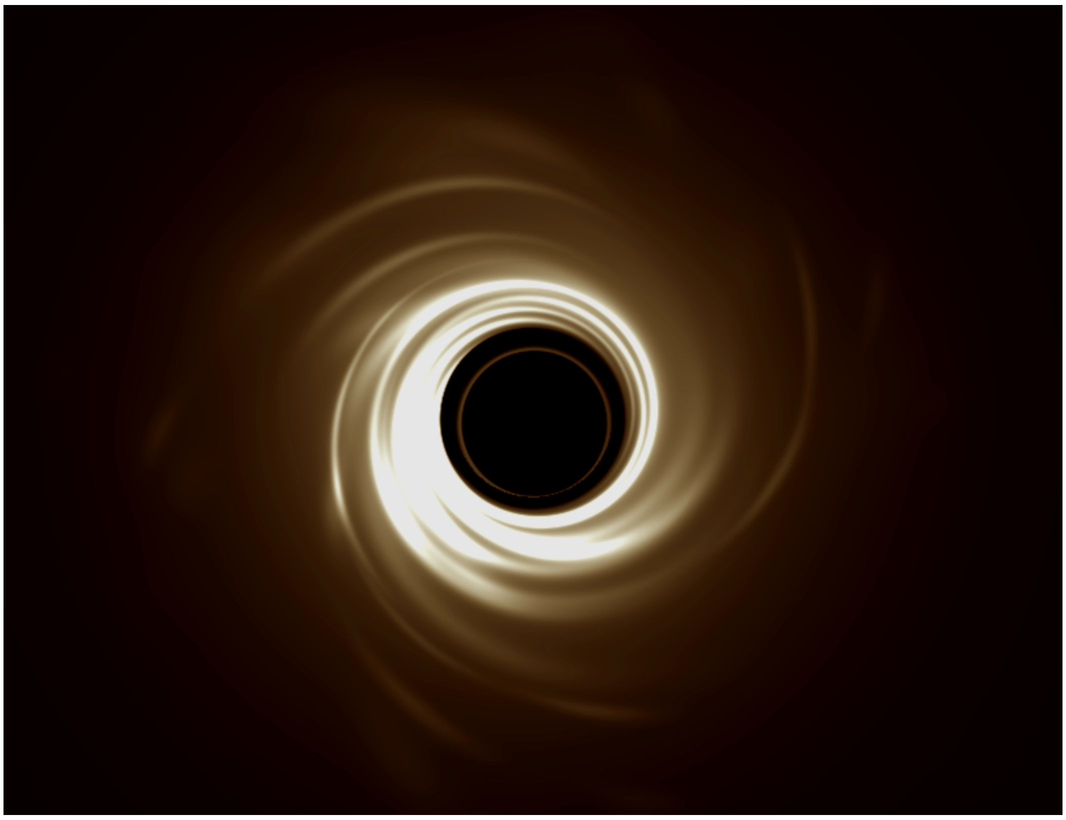

Figure 5.2. The view on the black from the top.

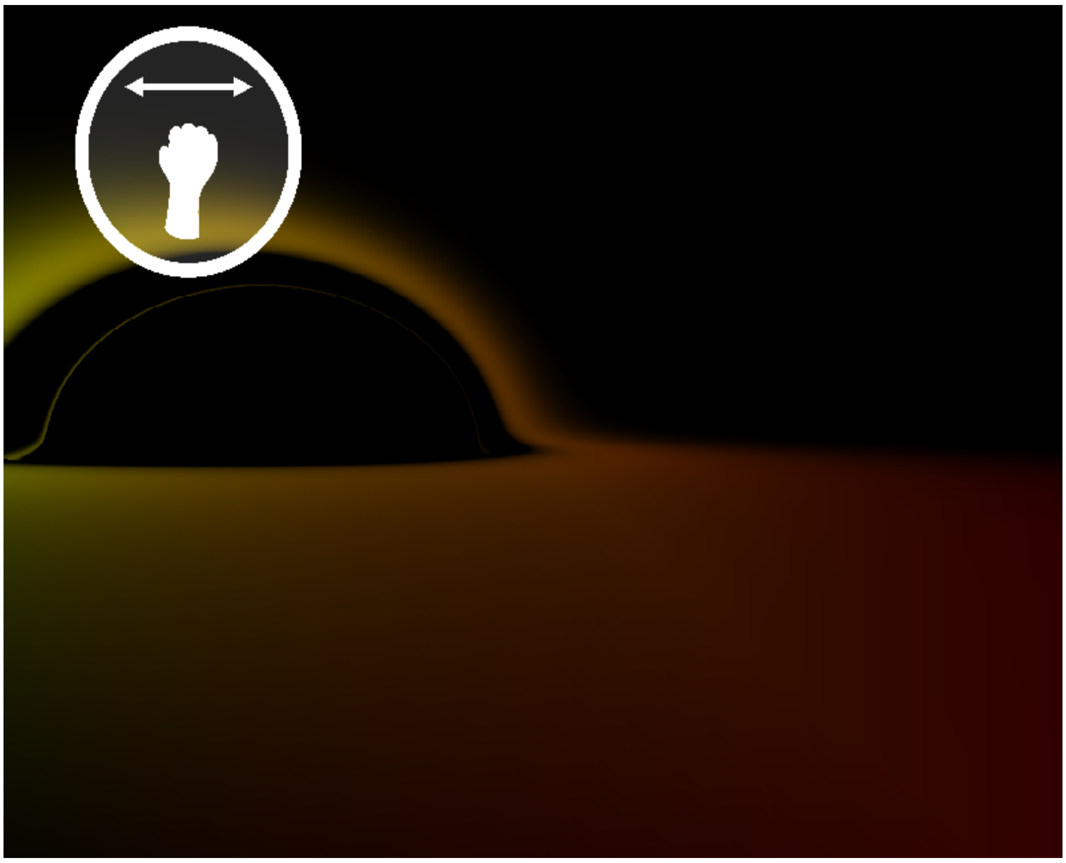

Figure 5.3. The view from the accretion disk with turned off light spots.

As we can see on the picture 5.3 the light spots are generating randomly around the black hole and then orbit it until the black hole absorbs them. From this point of view we can see that the light from the bottom side of the black hole is going to the top of it because of a gravity distortion. In this case the black hole works like a lens.

5.3 Usability tests

Our goal was to find usability problems and see what the real people who are not familiar with the black holes and relativistic physics will feel when they use the application. This section will describe the testing process, setup and the participants.

5.3.1 Setup

The testing was performed at the Triangle study room of the Faculty of Electrical Engineering of the Czech Technical University in Prague at the Department of Computer Graphics and Interaction. There were involved 3 participants in test. There was a PC with installed Ubuntu Linux distributive on it and all the necessary software that needed. On the desk were keyboard, mouse and Leap Motion controller already connected to the PC. Important note is that there were only participant and the instructor in the room, so nothing could disturb the participant from the testing process.

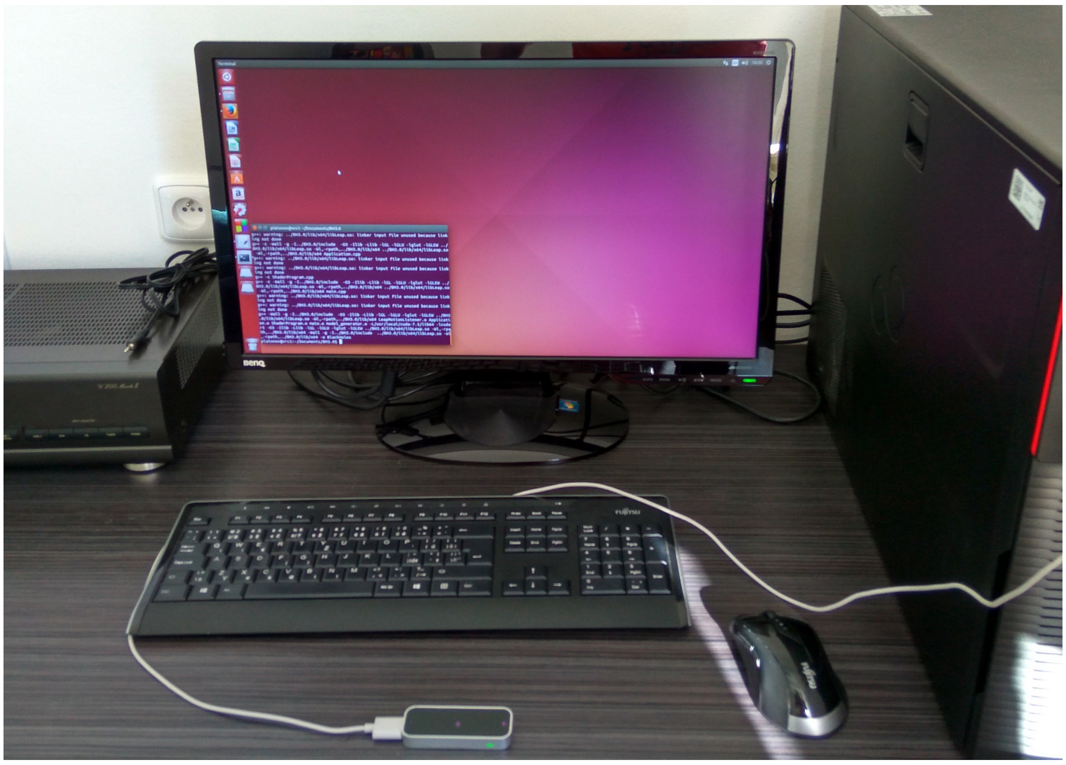

Figure 5.4. The setup for testing

5.3.2 Participants

All the participants are students of the Czech Technical University in Prague and all of them are about the same age (21-23). No one of the participants has ever seen the application before or even understood what the black hole is. The only knowledge that they have is that the black is a mysterious object in the universe. Also nobody have ever used the Leap Motion controller, so it was the first experience for the participants.

The first participant was a student of the Faculty of Transportation Sciences and as was already mentioned had no experience with the application. However, with good knowledges of physics the participant understood the basics of the nature of the black holes in a short period of time before testing.

The second participant was a student of the Faculty of Electrical Engineering and also had no experience with black holes. This participant has no knowledges of astrophysics and even had no clue about the gravity. So for him it was the very first introduction to this problem.

The third participant was also a student of the Faculty of Electrical Engineering and also had no knowledges about the black holes.

5.3.3 Questionnaire

The questionnaire contained 4 simple questions:

- What is your age?
- What is your education?
- Do you know anything about the black holes?
- Have you ever tried to interact with the hand tracking device?

Users completed it before the test. After the test we asked users another 5 simple questions:

- Did you like the application?
- Did you feel frustrated during the test?
- Have you learned something new about the black holes?
- The gestures that were used for interactions seemed for you intuitive and easy-to-use?
- Would you like to use this application again?

According to the user's answers we have created the overview of findings that is described in the next section.

5.3.4 Testing process and findings

After the short briefing and questionnaire all the participants had the basic knowledges about the black holes and gestures that they may use. The goal of the testing was to test if those gestures are intuitive and easy-to-use. We asked each participant to make the following actions:

- Zoom in the camera
- Rotate the camera left
- Zoom out the camera
- Change the camera's inclination
- Turn off the light spots
- Turn on the light spots
- Reset the scene

All of the participants had no problems with the gestures, but the second participant noticed that turning the light spots rotation should be continuously. In the current state of the application when this effect turns on or off, the light spots instantly appear or disappear. The first participant also noticed that for better understanding where the camera is situated there might be a background, that helps users to orientate in the world coordinates.

5.4 Benchmarking

In the following section we will create simple benchmark to compare the performance of the application at different resolutions and see if CUDA makes an improvement or not.

5.4.1 Benchmark concept

The concept is simple: setup application to start at needed resolution and add output to the text file after rendering each frame. This will show us how long it takes CPU or GPU to create a frame in real-time. For the ease of use we will use square aspect ratio. After the test we will find out an average values for further processing. To make the benchmarking more reliable we will use multiple setups with different configurations, that will allow us also to test the scalability of the application.

Technical specifications of the first PC that was used for benchmarking:

- CPU: Intel® Core™ i5-4690K (Haswell)
- Memory: 2x Kingston 8GB DDR3 1600MHz
- Video card: NVIDIA GeForce GT 730 (384 cuda cores)

Technical specifications of the second PC that was used for benchmarking:

- CPU: Intel® Xeon™
- Memory: 64GB DDR3 1600MHz
- Video card: NVIDIA GTX Titan (2688 cuda cores)

For easier recognition we will mark the first PC as *A* and the second as *B*.

5.4.2 Benchmark results

After the benchmarking we can create next comparison table 5.2 that shows average frame-rate at different resolutions for CPU and CUDA computations.

resolution	CPU@A	CUDA@A	CPU@B	CUDA@B
100x100	10	71	17	98
300x300	8	64	13	86
600x600	1	44	5	63
1000x1000	0.2	27	1	48

Table 5.2. Average frame-rate in frames per second for CPU and CUDA computations according to resolution.

5.4.3 Conclusion of benchmarking

As we can see from the previous subsection CUDA makes a dramatic improvement in the performance and provides us FPS that could be used in real-time visualization. However the complexity of the application remains high and at higher resolutions we can see a huge decreasing of FPS for both rendering methods. For sure there is needed an optimization for CUDA compatibility in the spots-generating part of the application.

Chapter 6
Conclusion

The goal of this project was to visualize physically accurate black hole model and add interactivity with help of motion controller and keyboard. This was achieved with help of Michal Bursa's library [3] and implementation of the application. The application provides easy-to-use, clear interface with controls and gestures that can be easily used even by kids. The application uses CUDA technology for better performance and provides excellent user experience. With help of technologies such as OpenGL, GLUT and GLEW the application remains cross-platform and stable.

The application supports Leap Motion controller, that can be easily connected to a USB port of a computer, without any necessary configurations. The application performs Kerr ray tracing and renders the output in real time without any delays at the resolution up to 720p. Light spots animation improves user experience and makes the application even more entertaining, also it can be easily turned off or on. All functional and non-functional requirements that are described in Chapter 3 have been completed. Leap Motion gestures that have been chosen seemed fluent and intuitive for testers. The benchmarking shows that CUDA technology provides much faster rendering time than rendering using CPU. The application remains cross platform and can be easily extended.

6.1 Future work

The next step in this project is further optimization of the CUDA part of the application and optimization inside the library. The results of the usability tests showed that there is still space for improvements, such as background, that could show user so-called ”gravity lens” effect and the light spots fluent turn off/on effect. The scale gesture recognition should be optimized to detect hands motions more precisely. Also the application GUI should be improved, for instance add the start guide that will describe the basics of the application and how to use gestures.

References

[1] Sean M. Carroll. *Lecture Notes on General Relativity*, December 1997.
http://preposterousuniverse.com/grnotes/grnotes-seven.pdf.

[2] Riccardo Antonelli *A real-time simulation of the visual appearance of a Schwarzschild Black Hole.*.
http://spiro.fisica.unipd.it/~antonell/schwarzschild/.

[3] Michal Bursa *Prague Relativistic Astrophysics.*
http://astro.cas.cz/michal-bursa.

[4] Luke Mastin *Event horizon and accretion disk.*
http://www.physicsoftheuniverse.com/topics_blackholes_event.html.

[5] Andrew Hamilton *More about the Schwarzschild Geometry*, February 2006.
http://casa.colorado.edu/~ajsh/schwp.html.

[6] Wolfram MathWorld *Aeronautical Terminology.*
http://mathworld.wolfram.com/Geodesic.html.

[7] Jason Dexter and Eric Agol *A fast new public code for computing photon orbits in a Kerr spacetime*, Department of Physics, University of Washington, Seattle, WA 98195-1560, USA; Department of Astronomy, University of Washington, Box 351580, Seattle, WA 98195, USA. 2009 April 27.

[8] M. Bursa *Raytracing in Kerr spacetime: correct formulae for azimuthal and time coordinates (Research Note)*, Astronomical Institute of the Czech Academy of Sciences (ASU), Bočnií II 1401/1, 141 00 Prague, CZ, 2016 May 4.

[9] S. Chandrasekhar *Mathematical theory of black holes*, Oxford University Press, 1983.

[10] TechTarget *Kinect.*
http://searchhealthit.techtarget.com/definition/Kinect.

[11] Leap Motion Developer *API Overview.*
https://developer.leapmotion.com/documentation/cpp/devguide/Leap_Overview.html.

[12] NVIDIA. *CUDA Parallel Computing Platform.*
http://www.nvidia.com/object/cuda_home_new.html.

[13] OpenGL. *OpenGL - The Industry's Foundation for High Performance Graphics.*
https://www.opengl.org/about/.

Appendix A
Abbreviations

The list of abbreviations that were used in this thesis.

A.1 Abbreviations

SM	Streaming Multiprocessors.
GUI	Graphical user interface.
GPU	Graphics processing unit.
CPU	Central processing unit.
SDK	Software development kit.
API	Application program interface.
FPS	Frames per second.
OS	Operating system.
CUDA	Compute Unified Device Architecture.
GLUT	The OpenGL Utility Toolkit.
GLEW	The OpenGL Extension Wrangler Library.
SOIL	Simple OpenGL Image Library.
GLSL	OpenGL Shading Language.
PC	Personal computer.

Appendix B
CD Contents

B.1 Software and libraries

- LeapMotionLibrary
- SOIL
- SIM5lib
- Executable binary
- Source files of the project
- Makefile to compile the sources

B.2 Text and documentation

- Project doxygen documentation (HTML and TEX versions)
- PDF version of the thesis
- TEX sources of the thesis
- README file with installation instructions

B.3 Media files

- Screenshots of the application