Mittuniversitetet
MID SWEDEN UNIVERSITY

Campus Härnösand Universitetsbacken 1, SE-871 88. **Campus Sundsvall** Holmgatan 10, SE-851 70 Sundsvall.
Campus Östersund Kunskapens väg 8, SE-831 25 Östersund.
Phone: +46 (0)771 97 50 00, Fax: +46 (0)771 97 50 01.

Introduction to Computer Graphics

Adam Temmel

Table of Contents

1 Introduction

Representing digital information has always been an interesting problem for science to solve, all the way back since the infancy of computing. Today, we have a diverse set of methods for representing computed data, ranging from simple terminals displaying lines of commands to complex rendering systems able to display realistic 3D-graphics. As rendering today has grown into becoming a moderately advanced topic, there exists quite a lot of wisdom to be found in this field.

1.1 Background and problem motivation

Given this, it would be of interest to make a deep dive into this particular field, in order to both get an understanding regarding the different subtopics covered within the actual topic as well as to apply this understanding in a practical manner. One way to accomplish this feat would be to look into a course for this topic.

1.2 Overall aim

The overall aim of this project is to by digesting material provided in a course (as well as other potential sources of knowledge) gain an increased understanding of computer graphics in some of its different forms.

1.3 Concrete and verifiable goals

The goals present for this project are as follows:

- The author should be able to present a basic understanding of some OpenGL/GLSL concepts.
- The author should be able to apply knowledge gained during the project regarding linear algebra within the field of computer graphics.
- The author should be able to apply some concepts attained from physics within the field of computer graphics.

1.4 Outline

Chapter 1 presents an introduction to the project. In chapter 2, some underlying theory is presented. Chapter 3 briefly describes the course that is followed throughout the project. Chapter 4 discusses some implementation details during the construction of the different components of the project. Chapter 5 showcases some screenshots of the results of the project, whereas chapter 6 rounds the report off with some discussion regarding the course and future work.

2 Theory

This chapter will present theory related to the project.

2.1 Graphics Processing Unit (GPU)

The GPU works as a complementary workhorse to the CPU. Whereas the CPU remains flexible in terms of what it is able to compute, the GPU is straight up a stronger version of a CPU when regarding raw computing power. The majority of GPUs enjoy working with triangles, partially due to it being a mathematically pleasant shape to work with, as well as it being a very flexible shape geometrically - most other shapes can be approximated by using the correct amount of triangles.[1]

2.2 OpenGL

The Open Graphics Library has been the most widely adopted graphics API within the industry for quite some time now, regardless of given the context of a 3D or 2D environment. The first version was released in 1992, with the latest version being released in 2017.[2]

2.2.1 Vertex Buffer Objects (VBO)

Vertex Buffer Objects is a method of passing vertex-related data to OpenGL. This is accomplished by first aligning all data of interest into one single block of contiguous memory, calculating the length of this memory block and then passing it to the appropriate OpenGL function. As OpenGL predominantly is a C-style API, it is also necessary to free this memory once the program is over.

```c
GLUint vbo;
const float data[] = {
    1.f, 0.f, 1.f, 0.f, 0.f, -1.f, -1.f, 0.f, 1.f
};

// create VBO
glGenBuffers(1, &vbo);

// set the VBO as active
glBindBuffer(GL_ARRAY_BUFFER, vbo);

// send vertex data to OpenGL
glBufferData(GL_ARRAY_BUFFER, sizeof(data), data, GL_STATIC_DRAW);

...

// cleanup
glDeleteBuffers(1, vbo);
```

Figure 1: Example of VBO usage within OpenGL.

2.2.2 Vertex Array Objects (VAO)

Vertex Array Objects are meant to be used together with VBOs, but instead of storing raw vertex data, VAOs store metadata related to how the VBO is constructed. Strides, offsets, as well as the amount of bytes provided per index are all valid examples of the type of attributes that are kept as information within a Vertex Array Object.

2.2.3 OpenGL Shading Language (GLSL)

The GLSL programming language is the language used to interact with and manipulate the graphics pipeline. The main benefit gained from using a specific language for this task is that developers no longer need to write hardware-specific or OS-specific code in order to program the GPU.[3]

2.3 Linear Algebra

Linear Algebra is the subset of mathematics which, amongst other things, discusses the presentation of different kinds of geometry and its transformations, including how geometry can be mapped from one location to another. Two central concepts to the field of linear algebra include the vector: an N-dimensional scalar, as well as the matrix: an $N \cdot M$-dimensional scalar.

2.3.1 Dot product

The dot product is an operation between two vectors (*a* and *b*) of equal dimension which produces a scalar. A more formal definition would look something like:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + ... a_n b_n$$

Figure 2: The definition of the dot product.

2.3.2 Cross product

The cross product is an operation between two vectors (*a* and *b*) of equal dimension which produces a third vector which is orthogonal to both prior vectors. A more formal definition would look something like:

$$a \times b = ||a||\,||b||\,sin(\theta)n$$

Figure 3: The definition of the cross product.

In *figure 3* θ represents the angle between *a* and *b* and *n* represents the unit vector of the plane which contains *a* and *b* in the direction which is given by the right hand rule. A more practical definition can be given if the dimension of the vectors is given, such as:[4]

$$u = (u_x, u_y, u_z), v = (v_x, v_y, v_z)$$
$$u \times v = ((u_y v_z - u_z v_y), (u_x v_z - u_z v_x), (u_x v_y - u_y v_x))$$

Figure 4: The second, sometimes more practical definition of the cross product.

2.3.3 Matrix Multiplication

For two matrices, A and B where :

$$A_{m,n} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix}$$

$$B_{n,m} = \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,m} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,m} \end{pmatrix}$$

Matrix multiplication is defined as the operation which:

$$C = A \cdot B = \begin{pmatrix} a_{1,1}b_{1,1} + \cdots + a_{1,n}b_{n,1} & a_{1,1}b_{1,2} + \cdots + a_{1,n}b_{n,2} & \cdots & a_{1,1}b_{1,m} + \cdots + a_{1,n}b_{n,m} \\ a_{2,1}b_{1,1} + \cdots + a_{2,n}b_{n,1} & a_{2,1}b_{1,2} + \cdots + a_{2,n}b_{n,2} & \cdots & a_{2,1}b_{1,m} + \cdots + a_{2,n}b_{n,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1}b_{1,1} + \cdots + a_{m,n}b_{n,1} & a_{m,1}b_{1,2} + \cdots + a_{m,n}b_{n,2} & \cdots & a_{m,1}b_{1,m} + \cdots + a_{m,n}b_{n,m} \end{pmatrix}$$

Figure 5: The formal definition of matrix multiplication

Matrix multiplication, pictured in *figure 5* is a way of expressing different transformations upon objects. These transformations are effectively chained by multiplying matrices together, and ultimately presented upon a point in space by taking a vector containing an origin point and transposing it so that it too may be multiplied with the product of the prior matrices.

3 Methodology / Model

This chapter aims to discuss the method used during the project.

3.1 Graphics Course

The course used throughout the project is split into four different blocks. The first block discusses vector math, the second block discusses matrix math, the third block discusses some fundamentals of OpenGL and the fourth block discusses basics regarding raytracing[5]. The course information is presented as a set of 8-15 minute long lectures, with a set of multiple-choice questions being presented at the end of each lecture.

4 Design / Implementation

As the course has a rather strict policy regarding the external distribution of source code which relates to the particular assignments mentioned in the course, this chapter will instead focus on discussing a couple of extra exercises which relate to the course material but were not presented as course-specific exercises.

4.1 Matrix sandbox

The first of these exercises is something that resembles a matrix sandbox of sorts, in which the user is meant to be presented an object as well as a set of parameters to manipulate this object. These parameters are as follows:

- Translation
- Scale
- Rotation

The crux here is that although these transformation usually are meant to be performed in a particular order, the user should be able to rearrange them in order to visualise what happens if this order is broken.

The first things constructed for this exercise were a couple of helper functions to assist in the relevant matrix math.

```
// create new matrix
function NewMat4(values) {
    if(values.length != 16) {
        throw "Length␣of␣values␣to␣construct␣matrix␣from␣must␣be␣16";
    }

    return [
            values.splice(0, 4),
            values.splice(0, 4),
            values.splice(0, 4),
            values.splice(0, 4),
        ];
}

// create new vector
function NewVec4(x, y, z, w) {
    return [x, y, z, w];
}

// matrix-matrix multiplication
function Mat4MultiplyMat4(lhs, rhs) {
    var result = Zero();
    for(var i = 0; i < 4; i++) {
        for(var j = 0; j < 4; j++) {
            for(var k = 0; k < 4; k++) {
                result[i][j] += lhs[i][k] * rhs[k][j];
            }
        }
    }
    return result;
}

// create new scale transform
function ScaleMat4(sx, sy, sz) {
    return NewMat4([
        sx, 0,  0,  0,
        0,  sy, 0,  0,
        0,  0,  sz, 0,
        0,  0,  0,  1,
    ]);
}

// create new translation transform
function TranslateMat4(tx, ty, tz) {
    return NewMat4([
        1, 0, 0, tx,
        0, 1, 0, ty,
        0, 0, 1, tz,
        0, 0, 0, 1,
    ]);
}
```

Figure 6: Some matrix related helper functions

```

// create new x-rotation transform
function RotateX(angle) {
    const c = Math.cos(angle);
    const s = Math.sin(angle);

    return NewMat4([
        1, 0,  0, 0,
        0, c, -s, 0,
        0, s,  c, 0,
        0, 0,  0, 1,
    ]);
}

// create new y-rotation transform
function RotateY(angle) {
    const c = Math.cos(angle);
    const s = Math.sin(angle);

    return NewMat4([
         c, 0, s, 0,
         0, 1, 0, 0,
        -s, 0, c, 0,
         0, 0, 0, 1,
    ]);
}

// create new z-rotation transform
function RotateZ(angle) {
    const c = Math.cos(angle);
    const s = Math.sin(angle);

    return NewMat4([
        c, -s, 0, 0,
        s,  c, 0, 0,
        0,  0, 1, 0,
        0,  0, 0, 1,
    ]);
}

function Rotate(x, y, z) {
    return Mat4MultiplyMat4(Mat4MultiplyMat4(RotateZ(z),
        RotateY(y)), RotateX(x));
}
```

Figure 7: Some rotation-related helper functions

These operations can be clustered a bit more, such as by authoring some sort of general transform object:

```
// create general transform object
function NewTransform(translate, scale, rotate) {
    return {
        translate: translate,
        scale: scale,
        rotate: rotate,
        rotateX: 0,
        rotateY: 0,
        rotateZ: 0,
    };
}

// compute a single matrix from a transform object
function DoRegularTransform(transform) {
    var result = Identity();
    result = Mat4MultiplyMat4(result, transform.translate);
    result = Mat4MultiplyMat4(result, transform.rotate);
    result = Mat4MultiplyMat4(result, transform.scale);
    return result;
};

// apply a transform matrix to a specific vector
function Mat4MultiplyVec4(mat, vec) {
    var result = NewVec4(0, 0, 0, 0)
    for(var i = 0; i < 4; i++) {
        for(var j = 0; j < 4; j++) {
            result[i] += mat[i][j] * vec[j];
        }
    }
    return result;
}
```

Now, all that remains is some way to draw lines from vectors. Luckily, this is not too difficult a task using an HTML canvas.

```
// find canvas in html
const canvas = document.getElementById("draw");
const ctx = canvas.getContext("2d");

// draw single line
function DrawLine(from, to) {
    ctx.beginPath();
    ctx.moveTo(from[0], from[1]);
    ctx.lineTo(to[0], to[1]);
    ctx.stroke();
}

// draw sequence of lines
function DrawPoints(points) {
    for(var i = 1; i < points.length; i++) {
        DrawLine(points[i - 1], points[i]);
    }
    DrawLine(points[points.length - 1], points[0]);
}
```

By then specifying a set of points, it is easy to define some sort of mesh. This mesh can be drawn using the `DrawPoints` function and transformed using the `DoRegularTransform` and `Mat4MultiplyVec4` functions in that order.

```
// create a set of points which corresponds to a grid
const grid = function() {
    const n = 20;
    var arr = new Array(n * 4);
    const half = n / 2;
    for(var i = 0; i < arr.length / 4; i++) {
        arr[i * 4] = NewVec4(i - half, -1, -half, 1);
        arr[i * 4 + 1] = NewVec4(i - half, -1, half, 1);
        arr[i * 4 + 2] = NewVec4(-half, -1, i - half, 1);
        arr[i * 4 + 3] = NewVec4(half, -1, i - half, 1);
    }

    arr.push(NewVec4(half, -1, half, 1));
    arr.push(NewVec4(-half, -1, half, 1));
    arr.push(NewVec4(half, -1, half, 1));
    arr.push(NewVec4(half, -1, -half, 1));

    return arr;
}();
```

```
// helper function to draw the aforementioned grid
function DrawGrid(points) {
    ctx.strokeStyle = "#aaaaaa";
    for(var i = 0; i < points.length; i++) {
        DrawLine(points[i], points[i + 1]);
        i++;
    }
    ctx.strokeStyle = "#000000";
}

// final function to transform and then draw the grid
function TransformAndDrawGrid() {
    var transformPre = DoRegularTransform(gridTransform);
    var transformPost = DoRegularTransform(cameraTransform);
    var transform = Mat4MultiplyMat4(transformPre, transformPost);
    var transformedGrid = Array.from(grid, point =>
        Mat4MultiplyVec4(transform, point));
    DrawGrid(transformedGrid);
}
```

Figure 8: Code showing what the final function calls to transform and draw a shape might look like

4.2 Raytracer

Raytracing is a method of rendering which sacrifices performance in order to draw a more authentic image. The main principle behind raytracing is quite simple: rays (of light) are being shot from a given camera position, with their journey being simulated until either the ray travels too far or if it stops hitting any of the shapes that are meant to be rendered. In fact, this procedure in itself, with all details left out can be summarized in just a few iterations, like in *figure 9*.

```
// create an array of Vec3
auto image = Image::create(1200, 720);

// for each pixel present
for(size_t x = 0; x < image.width; x++) {
    for(size_t y = 0; y < image.height; y++) {
        size_t yn = image.height - 1 - y;
        Vec3 color{};
        // for each sample that needs to be made
        for(size_t sample = 0; sample < samplesPerPixel;
            sample++) {
            // randomize ray direction
            auto u = float(x + randomFloat())
                / (image.width - 1);
            auto v = float(yn + randomFloat())
                / (image.height - 1);
            // create a ray
            auto ray = camera.ray(u, v);
            // calculate resulting color from ray
            color = color + rayColor(ray, world, maxDepth);

        }
        image(x, y) = normalizeColor(color, samplesPerPixel);
    }
}
```

Figure 9: A brief summary of the main algorithm behind most raytracers

The complications come from the `rayColor` function, which needs to iterate through all shapes that are being rendered in order to figure out if the ray has struck any of them. If that is the case, the material of this shape needs to be studied in order to both calculate the corresponding color gained from intersecting with the shape, but also in order to understand how the ray should behave from there on. Highly reflective materials will let the ray bounce again, thus needing a repetition of the process. As raytracing gains most of its realism from sampling a lot of rays, it also makes sense to introduce some randomness to these steps in order to better approximate the real world.

As an example for how the intersection might look like, *figure 10* shows some intersection code between a ray and a sphere.

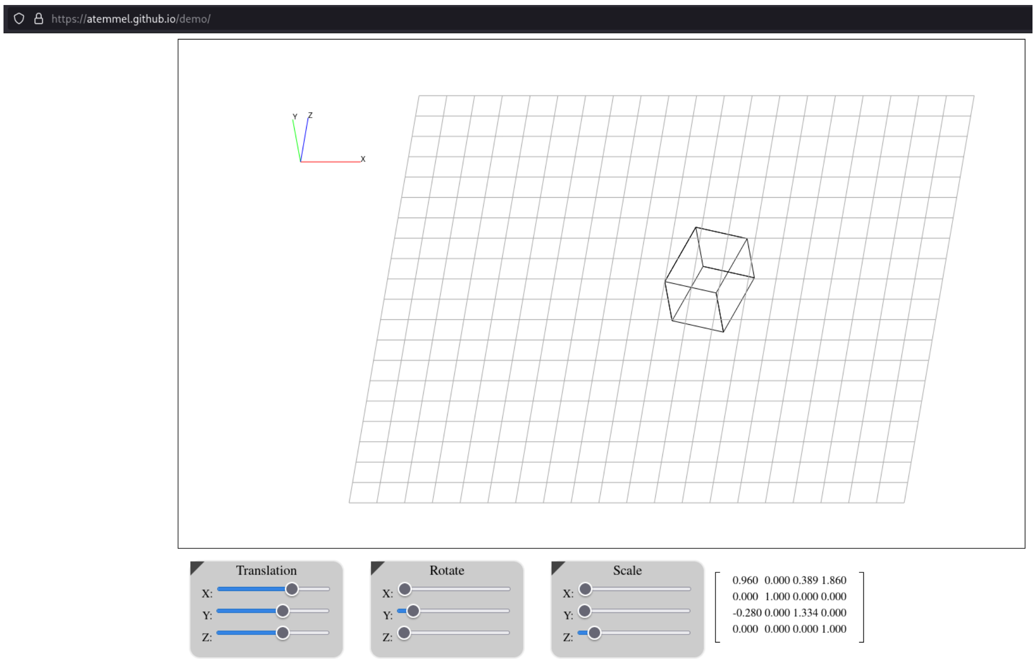
```cpp
auto Ray::hit(Sphere sphere, float minStep, float maxStep,
        RayHitData& data) const -> bool {
    // use a slightly more efficient version of the
    // pythagorean theorem for calculating the intersection
    auto oc = origin - sphere.origin;
    auto a = direction.squaredNorm();
    auto halfB = oc.dot(direction);
    auto c = oc.squaredNorm() - sphere.radius * sphere.radius;
    auto discriminant = halfB * halfB - a * c;

    // check to see if an intersection has occurred
    if(discriminant < 0.f) {
        return false;
    }
    auto sqrtDiscriminant = std::sqrt(discriminant);

    // same as before, but now with a range check
    auto root = (-halfB - sqrtDiscriminant) / a;
    if(root < minStep || maxStep < root) {
        root = (-halfB + sqrtDiscriminant) / a;
        if(root < minStep || maxStep < root) {
            return false;
        }
    }

    // collect relevant data for the intersection
    data.step = root;
    data.point = at(root);
    data.normal = (data.point - sphere.origin)
        / sphere.radius;
    data.material = sphere.material;
    return true;
}
```

Figure 10: Ray-Sphere intersection function

5 Results

This chapter intends to present the results of the project.

5.1 Matrix sandbox



```
0.960  0.000 0.389 1.860
0.000  1.000 0.000 0.000
-0.280 0.000 1.334 0.000
0.000  0.000 0.000 1.000
```

Figure 11: Screenshot taken from the matrix sandbox.

The matrix sandbox is presented in *figure 12* as an interactive website in which users are able to explore the possibilities of presenting transforms using matrices. The majority of the website shows a cube and a grid as well as a coordinate frame for the purpose of orienting the user. Also present is a set boxes containing sliders which represent individual properties that the different transformation matrices hold. These boxes can be rearranged by dragging and dropping them, resulting in new sets of transformations depending on the order that they are presented in. Lastly, the end transformation is visualized as a 4x4 matrix, allowing users to directly see the consequences of their transformations.

5.2 Raytracer

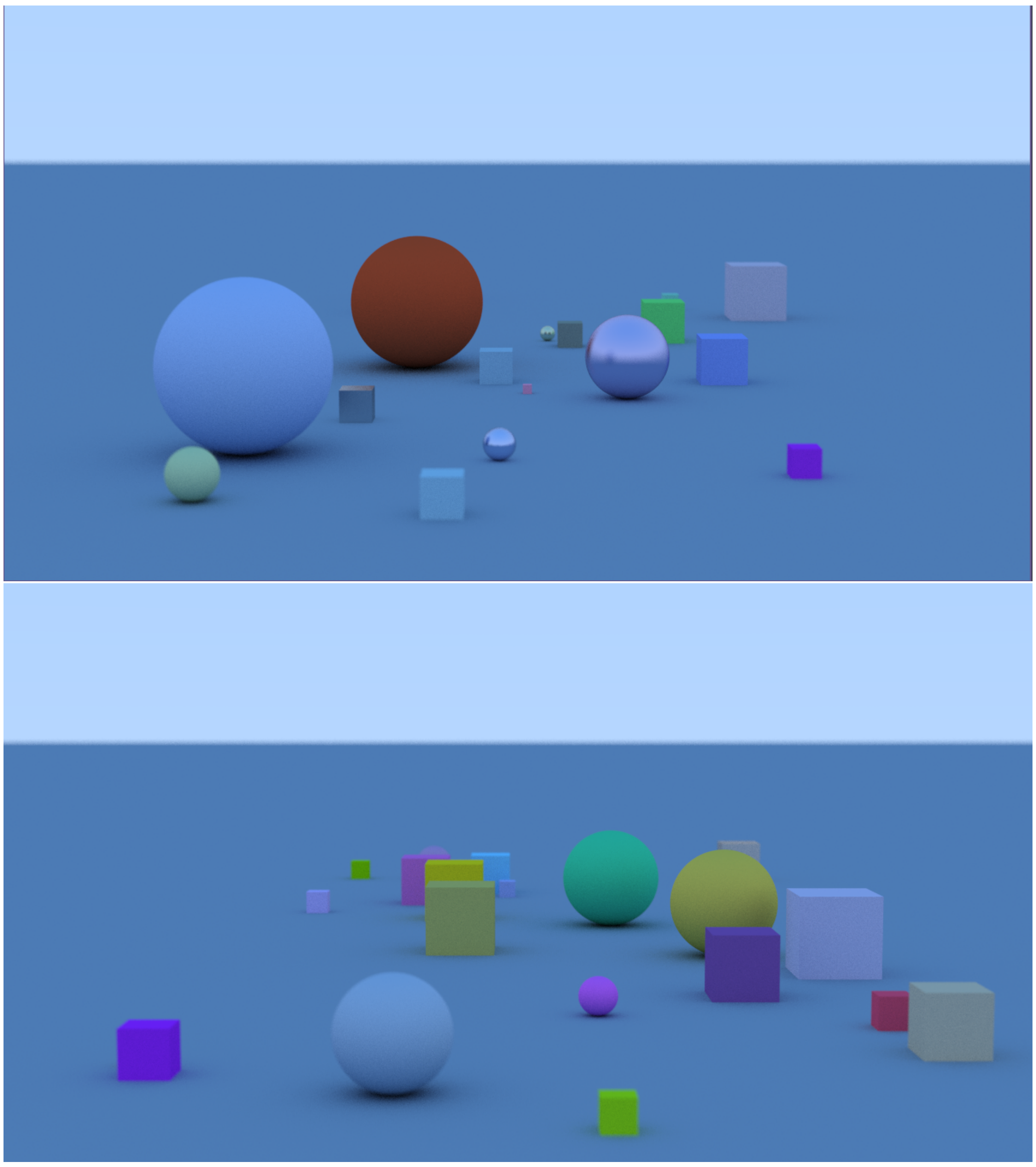


Figure 12: Screenshots taken from the raytracer.

The raytracer was also finalized, with features such as:

- Support for rendering of spheres and boxes.
- Support for lambertian and metallic materials.
- Multithreaded rendering.
- Basic depth-of-field effect.

Before the rendering starts, positions, dimensions and materials of several shapes are randomized, leading to each invocation of the raytracing producing a result that is likely to be completely different from the invocation before.

6 Discussion

Throughout both presented projects, a lot of linear algebra was necessary in order to finalize them. As such, it is safe to say that goal 2 is more than well achieved. Likewise, during the creation of the raytracer some understanding of physical concepts was also necessary to implement the different material properties. This satisfies the third goal. Lastly, the third course block presented some knowledge on the topic of OpenGL, which has briefly been summarized in the theory chapter, in turn completing the first goal.

6.1 Course discussion

The course was overall quite good, if albeit a little bit concise. Certain fields were not explained in incredible detail, but instead left for the student to figure out. This could very well have been an intentional decision from the authors of the course in order to ″force″ the student to grasp certain concepts. The foundation laid out during the course regarding certain math or graphics-related concept was rock-solid, however, making the course very suitable as an introduction to advanced computer graphics.

6.2 Future Work

A lot of different improvements can be done in the two projects, particularly the raytracer. Examples for improvements regarding the raytracer include, but are not limited to:

- Adding support for loading/saving scenes
- Adding support for other shapes than boxes and spheres.
- Adding support for different materials.
- Adding support for light sources within the scene.
- As well as further optimizing the general raytracing algorithm, as the current version of the raytracer takes several seconds to render a scene even with multithreading active.

References

[1] David Luebke and Greg Humphreys. In: *How Things Work* (2007), pp. 126, 130.

[2] The Khronos Group. *OpenGL Overview*. URL: `https://www.khronos.org/api/opengl` (visited on 01/15/2022).

[3] John Kessenich, Dave Baldwin, and Randi Rost. *The OpenGL Shading Language, Version 4.60.7*. 2019.

[4] Wolfram MathWorld. *Cross Product*. URL: `https://mathworld.wolfram.com/CrossProduct.html` (visited on 01/15/2022).

[5] EDX. *Computer Graphics Course*. URL: `https://www.edx.org/course/computer-graphics-2` (visited on 01/16/2022).