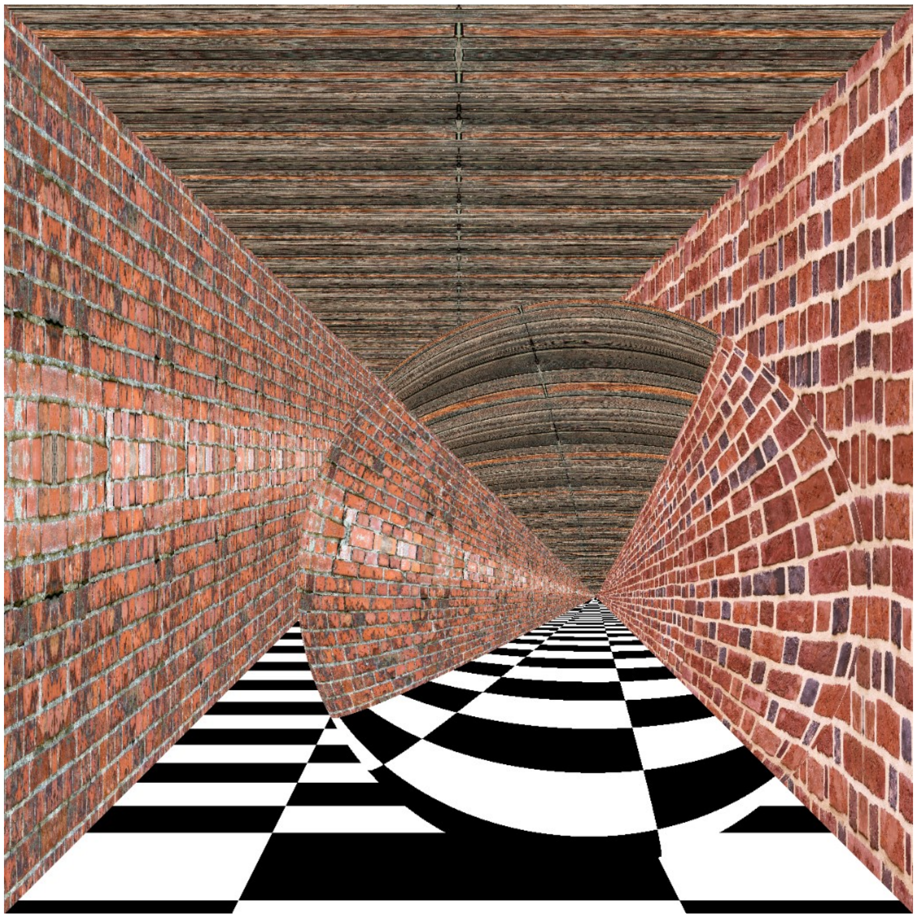
2015



Ray Tracing

RENDERING A PHOTOREALISTIC IMAGE OF A THREE DIMENSIONAL SCENE WITH THE USE OF RAY TRACING DESCRIBED BY VECTORS

FREDERIC J. TAUSCH

Table of content

Ray tracing

Rendering a photorealistic image of a three dimensional scene with the use of ray tracing described by vectors.

1 Introduction

Photorealistic rendering of three dimensional modeled scenes become more and more important during the last years. Besides the huge animation movie and gaming sector, engineers and designers are using photorealistic rendering of 3D modeled objects more and more often for different applications.[1]

Therefore, I decided to investigate how I can apply my knowledge about geometry in order to create a photorealistic image.

There are a lot of different engines out there to render three dimensional scenes, for example Cycles[2], Unreal Engine 4[3] or RenderMan[4]. However all these engines using mainly two different techniques to archive photorealistic images.[5]

The first one is the object-order based rendering technique, which starts by looking at each object.[6] The second one is the image-order based rendering technique, which starts by looking at each pixel of the image.[7] Both of these rendering techniques have their own advantages and disadvantages: for example, object-order based rendering is mostly faster and more suitable for real-time applications, contrary image-order rendering produces a more photorealistic image than object-order based rendering.[8] [9]

According to moore's law, the computing power grows each year[10], such that image-order rendering starts to get more suitable for real-time applications.

Due to these fact I will use ray tracing, an image-order base rendering technique, to archive a photorealistic image of a three dimensional scene.

[1] Executive summary about the animation and gaming industry in India (NASSCOM)
[2] Blenders powerful rendering engine Cycles (Blender Foundation)
[3] Unreal Engine 4 (Epic Games, Inc.)
[4] RenderMan a high performance renderer by Pixar (Pixar)
[5] Hidden Surface Algorithms – CSE 457 (University of Washington, Computer Science & Engineering)
[6] Ibidem
[7] Ibidem
[8] Ibidem
[9] Interactive Ray Tracing with CUDA (NVIDIA)
[10] Moor's Law and what it means (Kiel University, Faculty of Engineering)

2 THE IDEA BEHIND RAY TRACING

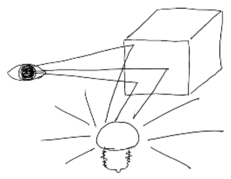


Figure 1- Visible Light

The main idea of ray tracing is to identify the color of each pixel by reconstructing the path that the light took, from the source of the light to the eye.[11] Therefore, we can create a photorealistic image with a perspective projection of the actual scene.

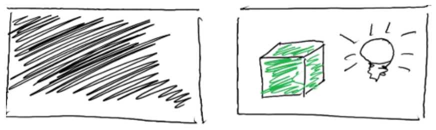


Figure 2- Green cube

For example, if we imagine a three-dimensional room with only a green cube in it, we would not see the cube, because the cube did not emit light by itself. Therefore, we have to add light source to see the green cube.
When we even go a little bit further and observe how the light reaches the retina of our eye, we could trace the path that a certain ray took, due to the corpuscular theory of light[12]. This path as shown in the figure 1 let us conclude that in order to see an object, the light travels from the source to an object and bounces off into our eye. The green color we see originates from the absorption spectrum of the green cube. Therefore, only green light bounces off and the cube absorbs the rest of the color spectrum.

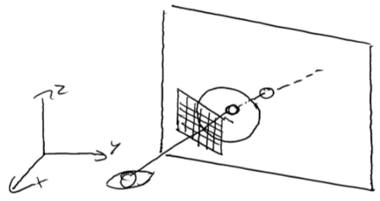


Figure 3 - Main idea of ray tracing

Since light travels in straight lines, according to the corpuscular theory of light[13], vectors could describe the path that a specific ray took. Due to the fact, that most of the emitted light does not hit our retina for this purpose it would not be efficient to calculate each emitted ray. A more efficient way is to calculate the ray recursive, from the eye to a possible source of light. When we use the information of the hit objects, we can conclude the color of the light. This method is known as ray tracing.

However, when we move from our imagined room into the real world we notice, that most of the light we see bounces around quiet often before it hits our retina. Additionally there are different surface of objects that have different effects on the path of the light, so it is very difficult to calculate the path each ray took. In order to deal with this complexity we have to make certain limitations that are described in the next chapter.

[11] Advanced Ray Tracing – Computer Graphics 15-462 (Carnegie Mellon University, School of Computer Science)
[12] Pierre Gassendi, Thomas Hobbes and Sir Isaac Newton have mostly developed the corpuscular theory of light. Since this theory only describes that light consists of small particles, it fails to describe effects as diffraction, interference and polarization. To get a correct simulation of the actual path of the light it would be necessary to use the quantum theory. Nevertheless, since we deal with objects relatively large in comparison to the wavelength of light the result would not differ in a large quantity. - Corpuscular theory of light (Wikipedia)
[13] Ibidem

3 IDEALIZED SETUP

As mentioned in a previous chapter, exact ray tracing is quiet complex. Therefore, in order to create a photorealistic image, I have to idealize my setup and look at only a few parts of Ray Tracing.

Therefore, I have made the following limitations to my setup:

- We consider an equal distributed global illumination of the whole scene as given, so we do not have to care about shadows and similar effects.
- The scene consist of two types of objects, planes and spheres.
  - These objects could have two types of surfaces, a mirror like surface (perfect reflection) or paper like surface (diffuse reflection).
  - The color of each point on the planes surface is defined by an image or a function that describes the texture of the surface.
  - A Sphere has the same color on every point of its surface.

From these limitations, I developed a scene that contains a mirror-sphere in the middle of an infinite long corridor with a chess texture on the floor.
To model this scene I used the following constellation of objects:

- To describe the corridor I used two pairs of parallel planes (E_1, E_2, E_3 and E_4) that are orthogonal to each other.
  - Every plane has a paper like surface.
  - One plane is defined as the floor of the corridor and is painted with chess pattern.
  - The other planes should have different textures to look more realistic.
- The camera is inside the corridor, looking towards a sphere with a mirror like surface.
  - The sphere should be inside the corridor.

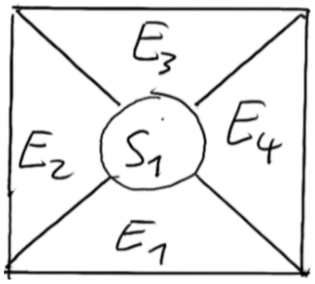


Figure 4 - Constellation ob objects in the scene

Nevertheless, it remains a quite complex setup, so I divided the problem into two major parts:

- Simple ray tracing covers only the basic functionality of ray tracing.
- Advanced ray tracing solves the advanced problems of perfect specular reflection and texture mapping.

Additional I created a Flow-Chart to describe the rendering process of the scene.

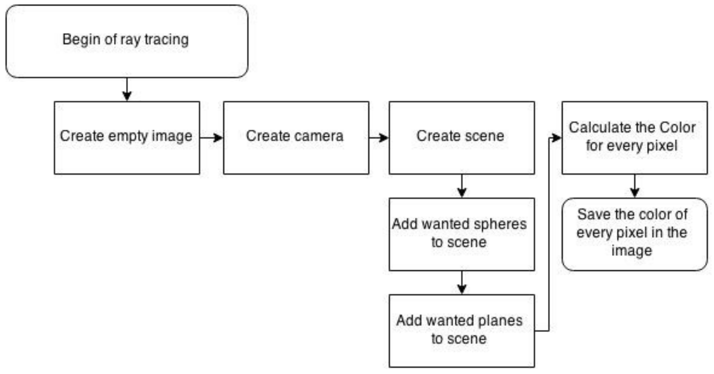

Begin of ray tracing
Create empty image
Create camera
Create scene
Add wanted spheres to scene
Add wanted planes to scene
Calculate the Color for every pixel
Save the color of every pixel in the image

Figure 5- Flow-Chart, process of rendering a scene

Since each of these parts is hard to deal with all at once, I divided the problems into smaller problems and solved them systematically (Divide and conquer).

4 USED TERMINOLOGY

After the rough overview on ray tracing, it is necessary to agree on some basic terminology before going into detail.

When a computer creates an image of different objects, also known as rendering, it should have a set of different objects in this picture. These objects are a part of the scene and have different attributes that define the properties of the object. Since a computer mostly performs ray tracing I will take a general approach on ray tracing to keep the result adaptable to different setups.

Since the aim of this exploration is to create a photorealistic image that extends the view through a screen into a virtual world, we have to create an imaginal Camera that takes this photo.
This camera should be the initial point of each traced ray and capture the light of the virtual world behind a screen. Given that the image sensor of a camera consist of a certain amount of small squares (a.k.a. pixel) that measure the color of the incident light, the resulting image consist as well of small pixels that represent the color of each square.[14]

In a later chapter, I will place a texture on the surface of an object, in order to add more detail. A texture is a normal bitmap with the same properties, typically a photo of the surface of an object.[15]

Additionally I want to mention that the $\cdot$ operation describes multiplication, $*$ describes the scalar product[16] and $\times$ describes the cross product[17].

[14] What is a pixel (WhatIs.com, TechTarget)
[15] Unity 3D manual, textures and videos (Unity Technologies)
[16] Dot Product (Wolfram MathWorld)
[17] Cross Product (Wolfram MathWorld)

5 RAY TRACING WITHOUT REFLECTING SURFACE

This chapter deals with the basic concept of ray tracing. Therefore, I try to reconstructing the path that the light took for each pixel and perform simple ray tracing for a more simplified scene.

5.1 PROCESS OF RAY TRACING

The process of ray tracing is very simple, and consist only of a few steps that are iterated for each pixel.

For each pixel:

1. Calculate the reversed vector of the ray, which goes through the pixel and the eye.
2. Find the first object hit by the ray.
3. Set the color of the pixel to the color of the object.

In order to tackle the problem I have divided the problem into two basic blocks, the modeling of the camera and the modeling of the scene. So the first step is solved with a description of the camera, when the rest of the steps will be tackled in the modeling of the scene.

5.2 MODELING OF THE CAMERA

Since we want to create a photo of scene behind a transparent screen, each ray of the scene has to go through the screen until it reaches our eye. Because of this, the color of each pixel in the photo represents the color of the ray that intersects with this pixel and the eye.
So in order to create a Camera and calculate each ray that went through a certain pixel, we have to define two things, the eye and the transparent screen.

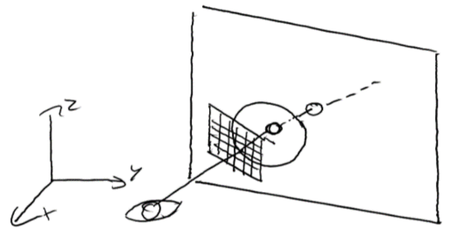


Figure 6 - Main idea of ray tracing

5.2.1 Define the camera

As mentioned before a camera consists of two different things, the eye and a transparent screen.
Let us define the position and orientation of the camera by the position and orientation of the eye. Therefore, we define the position of the eye by the position vector $\vec{e}$ of the point Eye, which is the origin of a new camera coordinate system. We describe every point in the camera coordinate system with three vectors $\vec{u}$, $\vec{v}$ and $\vec{w}$, that are orthogonal to each other. We place the transparent screen into this coordinate system and describe its position with its distance $\vec{d}$ to Eye and the orientation of the screen with a vector $\overrightarrow{\mathrm{viewUp}}$ that divides the screen vertically into two halves.

First of all, we define $\vec{u} = -\vec{d}$. Secondly we deduce from the knowledge that $\vec{v}$ is orthogonal to $\vec{u}$ and $\vec{v}$ is inside the plane which contains $\overrightarrow{viewUp}$ and $\vec{u}$, that $\vec{v} = \vec{u} \times \overrightarrow{viewUp}$. Due to the fact, that $\vec{u}$, $\vec{v}$ and $\vec{w}$ are orthogonal to each other, we can deduce $\vec{w} = \vec{u} \times \vec{v}$.

The next step is to normalize the vectors $\vec{v}$ and $\vec{w}$ to make it easier to calculate the position of each pixel on the screen. Since we are searching for the closes object behind the screen hit by the ray, it is easier to keep $\vec{u}$ as it is.

Lastly, we have to define the size of the screen. Therefore, we use the camera coordinate system to set the center of the screen to $\begin{pmatrix}-1\\0\\0\end{pmatrix}$ and define the boundaries of the screen by $b_{left} = \frac{-n_x}{2} \cdot pixel\ Size$, $b_{right} = \frac{n_x}{2} \cdot pixel\ Size$, $b_{bottom} = \frac{-n_y}{2} \cdot pixel\ Size$ and $b_{top} = \frac{n_y}{2} \cdot pixel\ Size$.

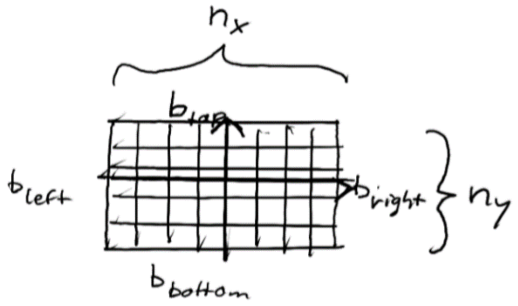


Figure 8 - Screen size

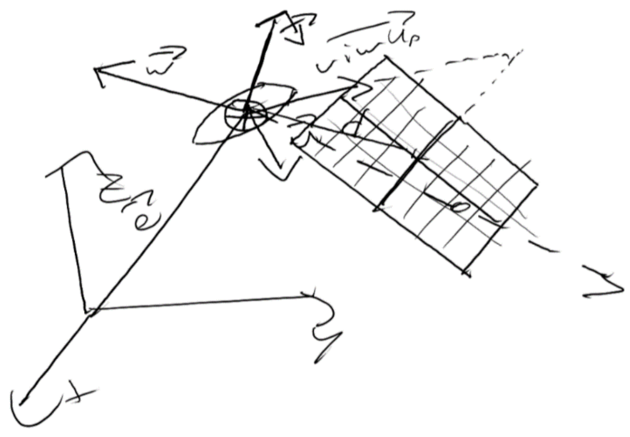


Figure 7 - Camera object

5.2.2 Tracing a ray

We start tracing the ray from the eye of the camera. From where we can define the position vector for the ray as $\overrightarrow{R(t)} = \vec{e} + t * \vec{d}$.

In order to calculate $\vec{d}$ we need the coordinates of the pixel (p_x , p_y) through which we want to trace the ray. We can deduce the vector describing the position of the pixel on the screen from the camera object. Since every point in the camera coordinate system could be described with $u \cdot \vec{u} + v \cdot \vec{v} + w \cdot \vec{w}$, we can describe the ray with $\overrightarrow{R(t)} = \vec{e} + t \cdot (u \cdot \vec{u} + v \cdot \vec{v} + w \cdot \vec{w})$.

In order to find u, v and w , we need to take a look on the definition of the screen in the camera object. From this definition, $= -|\vec{d}|$, $v = b_{left} + (b_{right} - b_{left}) \cdot \frac{p_x+0,5}{nx}$ and $w = b_{bottom} + (b_{top} - b_{bottom}) \cdot \frac{p_y+0,5}{ny}$ could be deduced.

$$\Rightarrow R(t): \vec{x} = \vec{e} + t \cdot (-|\vec{d}| \cdot \vec{u} + \left(b_{left} + (b_{right} - b_{left}) \cdot \frac{p_x+0,5}{nx}\right) \cdot \vec{v} + \left(b_{bottom} + (b_{top} - b_{bottom}) \cdot \frac{p_y+0,5}{ny}\right) \cdot \vec{w})$$

5.3 Modeling of the scene

Due to the limitations I created, the scene contains only two different types of objects, spheres and planes. Therefore, a mathematical description of these objects is needed to check if a ray hits the object. Since we only want to know if a defined ray hits the surface, implicit surface equations are a reasonable choice. Consequently, we can check if a specific point is on the surface of an object.

The implicit description of the surface of a sphere and a plane are derived in the following chapter.

5.3.1 Sphere

Let $\vec{c}$ be the position vector of the center C of a sphere with the radius r. For every point P on the surface of the sphere, the distance to the center C is r, for example $|\vec{p} - \vec{c}| = r$.[18]
It follows that $(\vec{p} - \vec{c}) * (\vec{p} - \vec{c}) = r^2$.

5.3.2 Plane

Let $\vec{n}$ be the surface normal of a plane E. Any point P in the plane E where $|\vec{n}| = 1$ and distance d to the origin can be described by $E(\vec{p}) = 0$ for $(\vec{p}) = \vec{n} * \vec{p} - d$.

5.4 Find intersections of rays and objects in the scene

In order to check if any ray intersects one of these objects we can use the implicit surface equation. Therefore, we just have to insert a ray into the previous defined implicit function and check if it equals zero for any > 1 , $t \in \mathbb{R}$.

If more than one object has an intersection with the ray for > 1 , we need to solve the equation for t, in order to find the first object hit by the ray behind the screen.

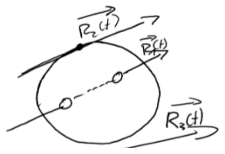
5.4.1 Sphere

In order to check if a ray R intersects the spheres surface, we have to calculate
$S_{3D\ Sphere}\left(\overrightarrow{R(t)}\right) = \left(\overrightarrow{R(t)} - \vec{c}\right) * \left(\overrightarrow{R(t)} - \vec{c}\right)$. If $S_{3D\ Sphere}\left(\overrightarrow{R(t)}\right) = r^2$ the ray will intersect the surface of the sphere.
If this is the case, the next step will be to solve $\left(\overrightarrow{R(t)} - \vec{c}\right) * \left(\overrightarrow{R(t)} - \vec{c}\right) = r^2$ for t, in order to find the first object hit by the ray behind the screen.

$$\left(\left(\vec{e} + t \cdot \vec{d}\right) - \vec{c}\right) * \left(\left(\vec{e} + t \cdot \vec{d}\right) - \vec{c}\right) - r^2 = 0$$

$$\Leftrightarrow \left(\vec{d} * \vec{d}\right) \cdot t^2 + \left(2 \cdot \vec{d}\right) * (\vec{e} - \vec{c}) \cdot t + (\vec{e} - \vec{c}) * (\vec{e} - \vec{c}) - r^2 = 0$$

Since this is a quadratic equation, it has two, one or zero solutions.
For $a = \vec{d} * \vec{d}$, $b = \left(2 \cdot \vec{d}\right) * (\vec{e} - \vec{c})$ and $c = (\vec{e} - \vec{c}) * (\vec{e} - \vec{c}) - r^2$ the solutions are
$t_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ for $b^2 - 4ac > 0$ or $t_1 = -\frac{b}{2a}$ for $b^2 - 4ac = 0$ or $\mathbb{L} = \{\}$ for $b^2 - 4ac < 0$, since $t \in \mathbb{R}$.



$\overrightarrow{R_2(t)}$
$\overrightarrow{R_1(t)}$
$\overrightarrow{R_3(t)}$

Figure 9 - Sphere intersections

[18] Implicit Differentiation (Massachusetts Institute of Technology)

5.4.2 Plane

If a ray R intersects a plane E, $E\left(\overrightarrow{R(t)}\right) = \vec{n} * \overrightarrow{R(t)} - d$ will be $\left(\overrightarrow{R(t)}\right) = 0$.

If the ray intersects a plane, the next step will be to solve $\vec{n} * \overrightarrow{R(t)} - d = 0$ for t, in order to find the first object hit by the ray behind the screen.

$$\vec{n} * \left(\vec{e} + t \cdot \vec{d}\right) - d = 0 \Leftrightarrow \left(\vec{d} * \vec{n}\right) \cdot t + \vec{e} * \vec{n} - d = 0$$

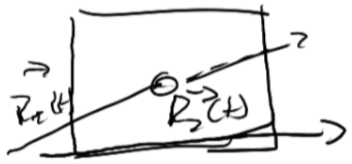


Figure 10 - Plane intersections

Since this is a linear equation, it has one or zero solutions.
Therefore $t = \frac{d - \vec{e} * \vec{n}}{\vec{d} * \vec{n}}$ for $\vec{n} * \vec{d} \neq 0$ or $\mathbb{L} = \{\,\}$ for $\vec{n} * \vec{d} = 0$.

5.5 Perform simple ray tracing

In order to check if my idea behind simple ray tracing is correct, I have written a little bit of Java code, as explained in a later section.
I used the following settings for a scene to perform simple ray tracing. Additionally a Color is assigned to each object, in order to color the pixel according to the first object hit by the ray of the pixel.

- Plane E_1 describes the floor
  $n_1 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$, $d_1 = -100$ and the color of E_1 is green
- Plane E_2 describes the left wall
  $n_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$, $d_2 = -100$ and the color of E_2 is red
- Plane E_3 describes the roof
  $n_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$, $d_3 = 100$ and the color of E_3 is yellow
- Plane E_4 describes the right wall
  $n_4 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$, $d_4 = 100$ and the color of E_4 is blue
- Sphere
  $\vec{c} = \begin{pmatrix} -600 \\ 0 \\ 0 \end{pmatrix}$, $r = 70$ and the color of the sphere is white
- Camera
  $\vec{e} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$, $\overrightarrow{\text{viewUp}} = \begin{pmatrix} -1 \\ 0 \\ 2 \end{pmatrix}$, $\vec{d} = \begin{pmatrix} -500 \\ 0 \\ 0 \end{pmatrix}$, $b_{left} = -112.5$, $b_{right} = 112.5$,
  $b_{bottom} = -112.5$, $b_{top} = 112.5$ and 2500 Pixel x 2500 Pixel
- Background is black

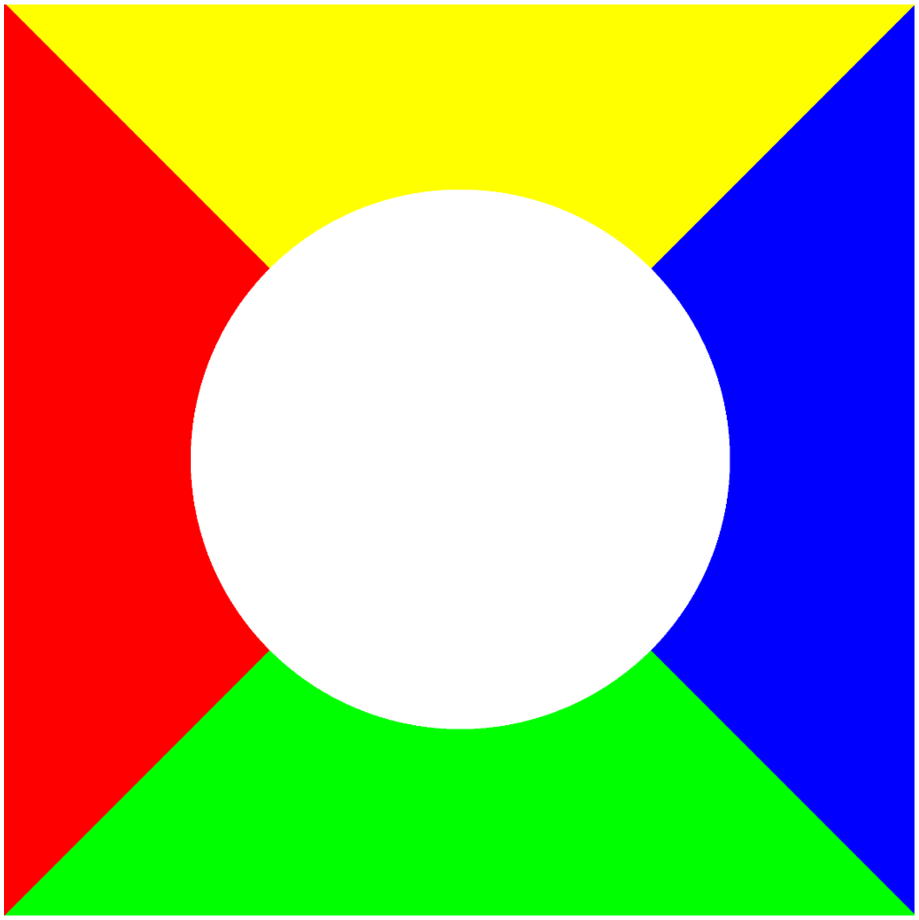


Figure 11 - Rendered image, ray tracing without reflection

The image has a resolution of 2500 pixel x 2500 pixel and has been rendered in 3 seconds on an Intel Xeon E3 1230 V3 processor (3.7 GHz).

It is great to see that my considerations working correctly, even if the picture is not looking photorealistic. Furthermore, it is a perfectly rendered image of my simple setup. The resulting image is a perspective view on the whole scene, consisting of the four walls and a sphere between them. Nevertheless, the image did not look photorealistic, because the color of the objects look unnaturally and shading, reflection or other effects are not visible.
Additionally the rendering time for the image was very high, because I have to calculate the color of $625 \cdot 10^4$ pixels.

In order to improve the impression of the rendered image, I will implement the further functionalities reflection and texture mapping, so it will look more photorealistic.

6 Ray tracing with perfect reflection

Advanced ray tracing covers perfect specular reflection and texture mapping.

6.1 Calculate reflection on rays

In order to calculate the reflected rays on any surface, we have to reduce the problem to a much simpler situation, so we can get an idea how this is done.
If we consider a line, in two-dimensional space and reflect a vector $\vec{r}$ at a point P on the line, we get the resulting vector $\overrightarrow{r'}$.

According to shells law[19] $\angle(\vec{n},\vec{r}) = \angle\left(\overrightarrow{r'},\vec{n}\right)$, where $\vec{n}$ is the surface normal of the line and points in the opposite direction of $\vec{r}$. In order to solve this equation geometrically for $\overrightarrow{r'}$ we use the trigonometric properties of the dot product.

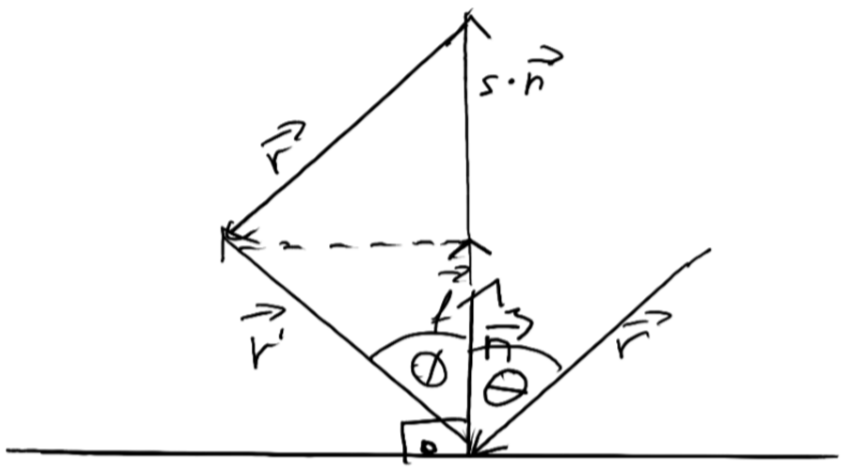


Figure 12 - Reflection

We consider a parallelogram where one of each pair of parallel sides are described with $\overrightarrow{r'}$ and $\vec{r}$.
Additionally $s \cdot \vec{n}$ for $s \in \mathbb{R}$ is a diagonal of the parallelogram.
From the geometric properties of a parallelogram, we can deduce that $\cdot\, \vec{n} + \vec{r} = \overrightarrow{r'}$.
Since s is unknown, we need to find another way to describe $\cdot\, \vec{n}$. Therefore we use vector projection, in order to describe $s * \vec{n}$ with a projection of the vector $-\vec{r}$ on $\vec{n}$.
Let $\vec{f}$ be the projection of the vector $-\vec{r}$ on $\vec{n}$, then $2 \cdot \vec{f} = s \cdot \vec{n}$ for $\vec{f} = \frac{\vec{n}*(-\vec{r})}{|\vec{n}|*|\vec{n}|} \cdot \vec{n}$.
Since we use a normalized form of the vector $\vec{n}$ we can simplify $\vec{f} = \vec{n} * (-\vec{r}) \cdot \vec{n}$.
From this we can deduce that that $\overrightarrow{r'} = 2 \cdot \vec{n} * (-\vec{r}) \cdot \vec{n} + \vec{r}$.

If we want to reflect a ray on a spheres surface, we have to get $\vec{n}$ for any point P on the surface of the sphere. From the knowledge about a sphere, we can deduce that $\vec{n} = \vec{c} - \vec{p}$.

[19] Dr. K. A. Tsokos: Physics for IB Diploma, Cambridge 2010, page 231-232

6.2 TEXTURE MAPPING

In order to find the color of a point P on the surface of a plane, we need to know were the point P is in relation to a fixed-point Z on the plane, so we can map the texture on the surface of the plane.

6.2.1 Map a point on the planes surface

To define the coordinate system of the plane we use a similar approach as described in 5.2.1 “Define the camera”. We describe every point in the planes coordinate system with two orthogonal vectors $\vec{u}$ and $\vec{v}$. In order to define $\vec{u}$ and $\vec{v}$ we need to define a point Z as the origin and a rotation vector $\overrightarrow{rot}$ to describe the rotation and orientation of the texture. From this we can define $\vec{u} = \vec{n} \times \overrightarrow{rot}$ and $\vec{v} = \vec{n} \times \vec{u}$.
Since any point P on the planes surface could be described with $\vec{p} = x \cdot \vec{u} + y \cdot \vec{v} \mid x, y \in \mathbb{R}$, because $\vec{u}$ and $\vec{v}$ are orthogonal. We need to solve this system of linear equations for x and y in order to get the x and y coordinates on the surface of the plane. In order to solve this system of linear equations I use the Gaussian elimination.

The resulting coordinates can be used to get the color of the point P by checking the color of the same coordinates in the texture.

6.2.2 Define textures

In order to map the textures on a plane, textures need to have certain properties, such as an infinite size or a structure based on a Cartesian coordinate system. Therefore, we need to adjust images and functions in order to fit as a texture.

6.2.2.1 Chess pattern

In order to describe the chess pattern I have to develop a function, which describes whether a pixel is black or white.

Therefore, we need to find a function that describes the color of each tile by their x and y coordinates. Since a chess pattern consist of a basic pattern of two tiles iterated repeatedly, we should try to define the color of each tile by locking at the basic case.

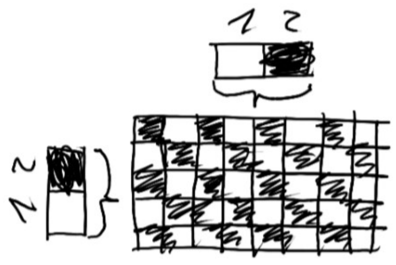


Figure 13 - chess pattern

Because there are only two tiles, the number of each tile could be even or odd. So if the number of each tile is even or odd determines the color of each tile. Since the chess pattern is not only one dimensional, but two dimensional, the basic pattern is ordered horizontally and vertically. Therefore we need to consider the sum of the integer x and y coordinates and look whether the sum is odd or even. So because of this the color could be described with : $(x, y) = \begin{Bmatrix} black, if\ (\lfloor x \rfloor + \lfloor y \rfloor)\ even \\ white, if\ (\lfloor x \rfloor + \lfloor y \rfloor)\ odd \end{Bmatrix}$.

6.2.2.2 Image as texture

Since the properties of an image did not fit to the demands of the texture, because the size of an image is limited, we need to loop the image in order to create a texture.
Therefore we compare the coordinates x and y with the $width$ and $height$ of the image when $x > width$ or $y > height$ we subtract $width$ from x or $height$ from y. Afterwards we iterate the whole process until $x < width$ and $< height$.
This can be simplified by “Get the color of the pixel $(x\ mod\ width, y\ mod\ height)$“, in order to create a looped image as a texture.

6.2.3 Resulting ray tracing process

I created a Flow-Chart in order to describe the final ray tracing process.

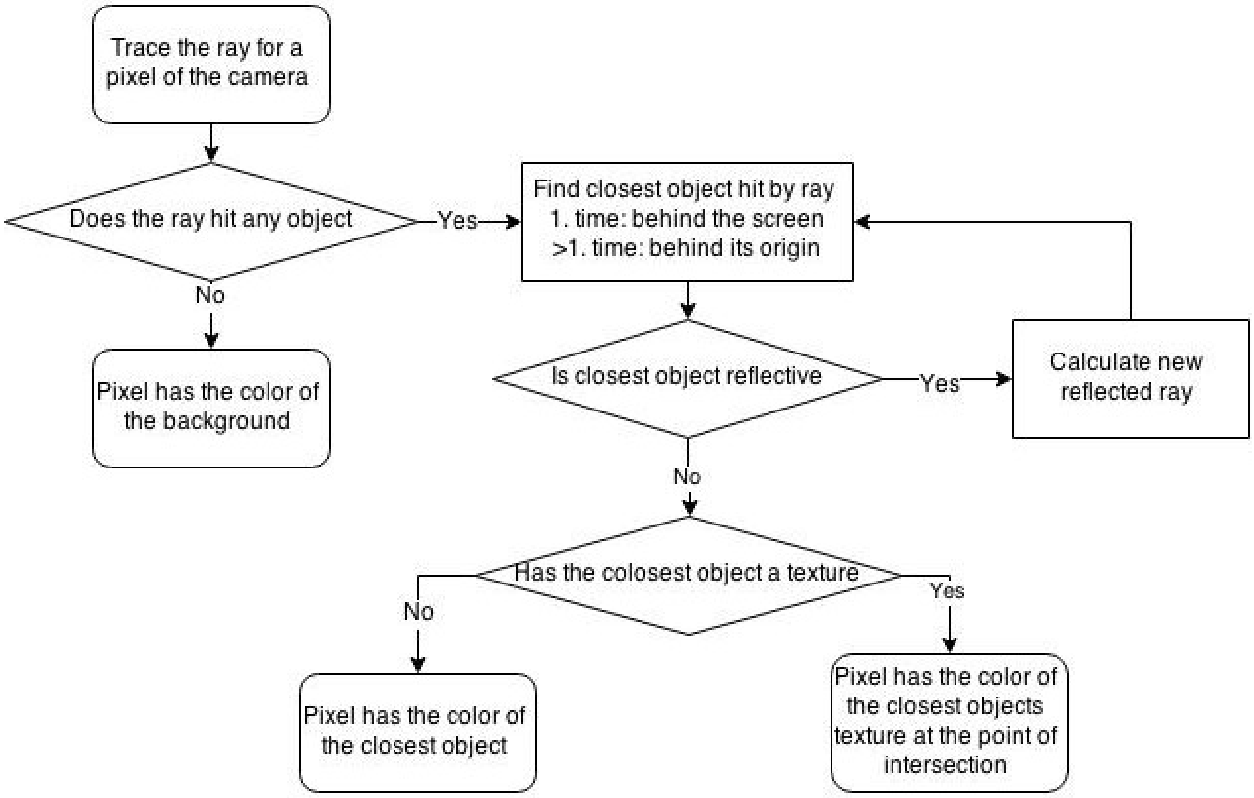

Trace the ray for a pixel of the camera
Does the ray hit any object
Yes
No
Pixel has the color of the background
Find closest object hit by ray
1. time: behind the screen
>1. time: behind its origin
Is closest object reflective
Yes
Calculate new reflected ray
No
Has the colosest object a texture
No
Yes
Pixel has the color of the closest object
Pixel has the color of the closest objects texture at the point of intersection

Figure 14 - Flow-Chart, final ray tracing process

7 IMPLEMENTATION INTO JAVA

My aim for the implementation of the previous described solutions was to see how well my methods of ray tracing performed. Additionally I wanted to expand my knowledge in coding and develop my own code entirely from the ideas I had. Therefore, I tried to write as much as I could from the ground up and minimize the dependence on other work.
The main approach was dividing into different objects according to the mathematical ideas named in previous parts. So the code is written object oriented with a direct transfer of the used mathematics.

The developed code only depends on the libraries of the Java development kit[20] and a class for the Gaussian elimination by Sebastian Peuser[21]. Additionally I found some inspiration for the vector class from Leonard McMillans[22] work. The rest was done by myself.

7.1 STRUCTURE

The UML diagram bellow describes the used classes:

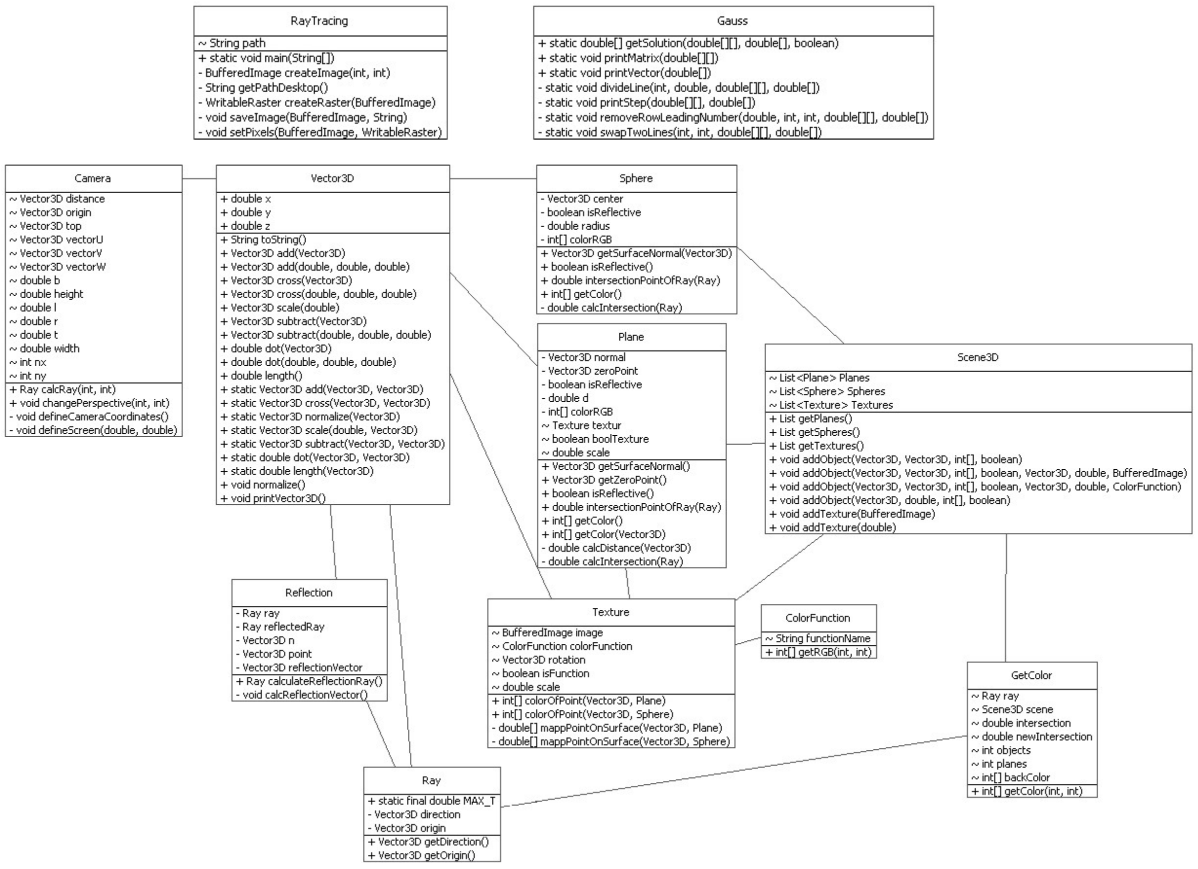

RayTracing
~ String path
+ static void main(String[])
- BufferedImage createImage(int, int)
- String getPathDesktop()
- WritableRaster createRaster(BufferedImage)
- void saveImage(BufferedImage, String)
- void setPixels(BufferedImage, WritableRaster)
Gauss
+ static double[] getSolution(double[][], double[], boolean)
+ static void printMatrix(double[][])
+ static void printVector(double[])
- static void divideLine(int, double, double[][], double[])
- static void printStep(double[][], double[])
- static void removeRowLeadingNumber(double, int, int, double[][], double[])
- static void swapTwoLines(int, int, double[][], double[])
Camera
~ Vector3D distance
~ Vector3D origin
~ Vector3D top
~ Vector3D vectorU
~ Vector3D vectorV
~ Vector3D vectorW
~ double b
~ double height
~ double l
~ double r
~ double t
~ double width
~ int nx
~ int ny
+ Ray calcRay(int, int)
+ void changePerspective(int, int)
- void defineCameraCoordinates()
- void defineScreen(double, double)
Vector3D
+ double x
+ double y
+ double z
+ String toString()
+ Vector3D add(Vector3D)
+ Vector3D add(double, double, double)
+ Vector3D cross(Vector3D)
+ Vector3D cross(double, double, double)
+ Vector3D scale(double)
+ Vector3D subtract(Vector3D)
+ Vector3D subtract(double, double, double)
+ double dot(Vector3D)
+ double dot(double, double, double)
+ double length()
+ static Vector3D add(Vector3D, Vector3D)
+ static Vector3D cross(Vector3D, Vector3D)
+ static Vector3D normalize(Vector3D)
+ static Vector3D scale(double, Vector3D)
+ static Vector3D subtract(Vector3D, Vector3D)
+ static double dot(Vector3D, Vector3D)
+ static double length(Vector3D)
+ void normalize()
+ void printVector3D()
Sphere
- Vector3D center
- boolean isReflective
- double radius
- int[] colorRGB
+ Vector3D getSurfaceNormal(Vector3D)
+ boolean isReflective()
+ double intersectionPointOfRay(Ray)
+ int[] getColor()
- double calcIntersection(Ray)
Plane
- Vector3D normal
- Vector3D zeroPoint
- boolean isReflective
- double d
- int[] colorRGB
~ Texture textur
~ boolean boolTexture
~ double scale
+ Vector3D getSurfaceNormal()
+ Vector3D getZeroPoint()
+ boolean isReflective()
+ double intersectionPointOfRay(Ray)
+ int[] getColor()
+ int[] getColor(Vector3D)
- double calcDistance(Vector3D)
- double calcIntersection(Ray)
Scene3D
~ List<Plane> Planes
~ List<Sphere> Spheres
~ List<Texture> Textures
+ List getPlanes()
+ List getSpheres()
+ List getTextures()
+ void addObject(Vector3D, Vector3D, int[], boolean)
+ void addObject(Vector3D, Vector3D, int[], boolean, Vector3D, double, BufferedImage)
+ void addObject(Vector3D, Vector3D, int[], boolean, Vector3D, double, ColorFunction)
+ void addObject(Vector3D, double, int[], boolean)
+ void addTexture(BufferedImage)
+ void addTexture(double)
Reflection
- Ray ray
- Ray reflectedRay
- Vector3D n
- Vector3D point
- Vector3D reflectionVector
+ Ray calculateReflectionRay()
- void calcReflectionVector()
Texture
~ BufferedImage image
~ ColorFunction colorFunction
~ Vector3D rotation
~ boolean isFunction
~ double scale
+ int[] colorOfPoint(Vector3D, Plane)
+ int[] colorOfPoint(Vector3D, Sphere)
- double[] mappPointOnSurface(Vector3D, Plane)
- double[] mappPointOnSurface(Vector3D, Sphere)
ColorFunction
~ String functionName
+ int[] getRGB(int, int)
GetColor
~ Ray ray
~ Scene3D scene
~ double intersection
~ double newIntersection
~ int objects
~ int planes
~ int[] backColor
+ int[] getColor(int, int)
Ray
+ static final double MAX_T
- Vector3D direction
- Vector3D origin
+ Vector3D getDirection()
+ Vector3D getOrigin()

Figure 15 - UML diagram

[20] Java 8 documentation (Oracle)
[21] Gauss elimination implementation (Sebastian Peuser)
[22] Instructional Ray-Racing Renderer – only Vector class has been used (Leonard McMillan, MIT 6.837 Fall -98)

7.2 Result

A similar setup as in 5.5 “Perform simple ray tracing” defined was used to render the picture on the cover sheet. Only the position of the sphere and the surfaces of the objects have been modified.
It took an Intel Xeon E3 1230 V3 processor (3.7 GHz) 992 minutes and 5 seconds to render the 2500 pixel x 2500 pixel image.
In order to show the benefits of my universal approach on ray tracing, I want to show you the results of the render process from a different point of view. So I rotated the camera by an angle of 45° on the X-Y Plane and changed its position.
Therefore, I used the following settings:

- Plane E_1 describes the floor
  $n_1 = \begin{pmatrix}0\\0\\1\end{pmatrix}$, $d_1 = -100$, the texture of E_1 is the chess function
  $\vec{z}_1 = \begin{pmatrix}-600\\0\\100\end{pmatrix}$ and $\overrightarrow{rot}_1 = \begin{pmatrix}0\\1\\1\end{pmatrix}$
- Plane E_2 describes the left wall
  $n_2 = \begin{pmatrix}0\\1\\0\end{pmatrix}$, $d_2 = -100$, the texture of E_2 is an aged brick wall
  $\vec{z}_2 = \begin{pmatrix}-600\\-100\\0\end{pmatrix}$ and $\overrightarrow{rot}_2 = \begin{pmatrix}0\\1\\1\end{pmatrix}$
- Plane E_3 describes the roof
  $n_3 = \begin{pmatrix}0\\0\\1\end{pmatrix}$, $d_3 = 100$, the texture of E_3 is a wooden ceiling
  $\vec{z}_3 = \begin{pmatrix}-600\\0\\-100\end{pmatrix}$ and $\overrightarrow{rot}_3 = \begin{pmatrix}-1\\0\\-1\end{pmatrix}$
- Plane E_4 describes the right wall
  $n_4 = \begin{pmatrix}0\\1\\0\end{pmatrix}$, $d_4 = 100$, the texture of E_4 is a new brick wall
  $\vec{z}_4 = \begin{pmatrix}-600\\100\\0\end{pmatrix}$ and $\overrightarrow{rot}_4 = \begin{pmatrix}0\\-1\\1\end{pmatrix}$
- Sphere
  $\vec{c} = \begin{pmatrix}-600\\100\\30\end{pmatrix}, = 70$, the color of the sphere is silver and the sphere is reflective
- Camera
  $\vec{e} = \begin{pmatrix}-146.45\\-353.55\\0\end{pmatrix}$, $\overrightarrow{\text{viewUp}} = \begin{pmatrix}-1\\1\\2\end{pmatrix}$, $\vec{d} = \begin{pmatrix}-353.55\\353.55\\0\end{pmatrix}$, $b_{left} = -112.5$, $b_{right} = 112.5$, $b_{bottom} = -112.5$, $b_{top} = 112.5$ and 2500 Pixel x 2500 Pixel
- Background is black

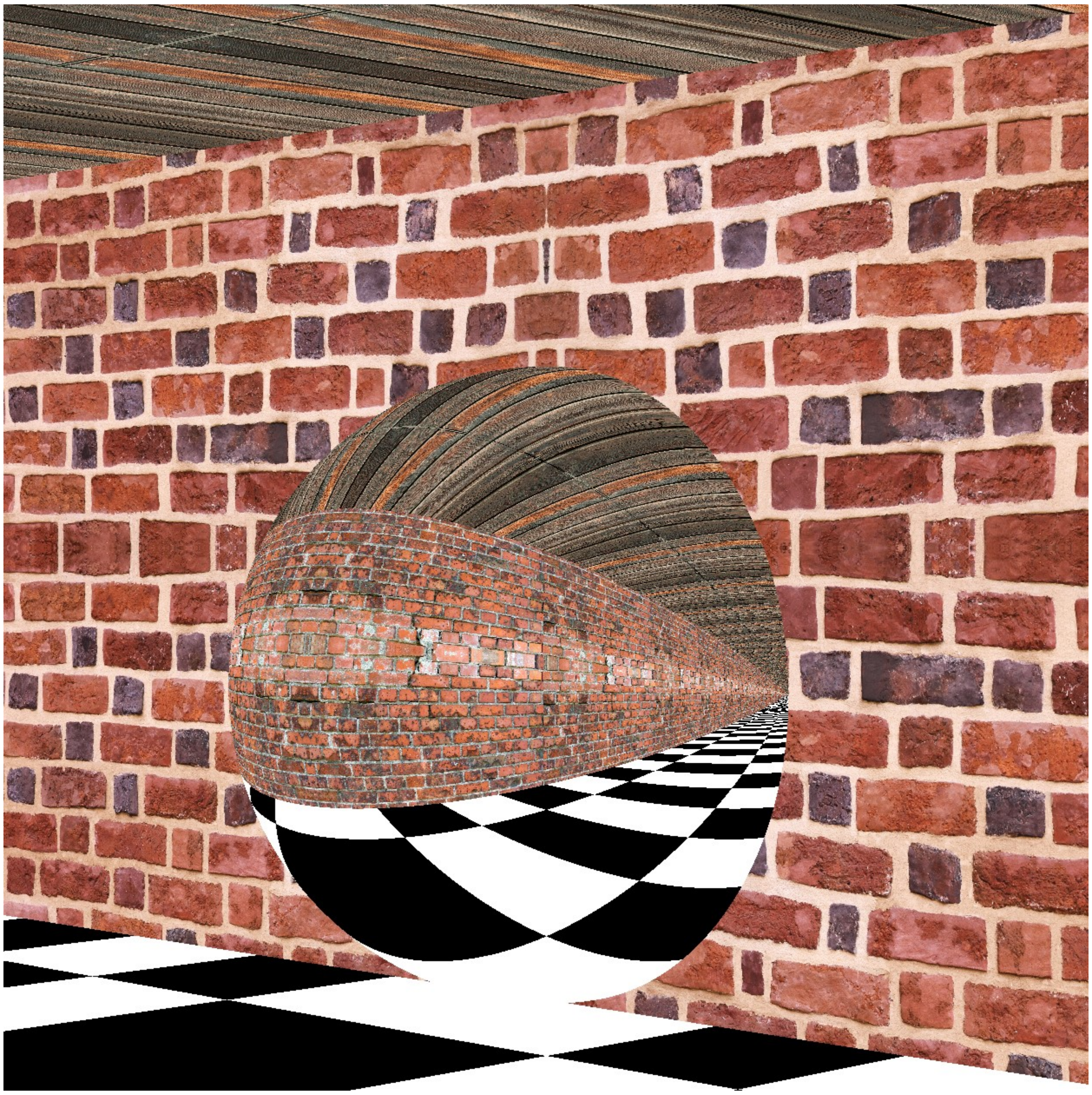


Figure 16 - Rendered image, inal result

The image has a resolution of 2500 pixel x 2500 pixel and has been rendered in 4 seconds on an Intel Xeon E3 1230 V3 processor (3.7 GHz).

8 CONCLUSION AND REFLECTION

During the mathematical exploration, I discovered how important ray tracing in our modern world is. Furthermore, ray tracing describes a mathematical problem that is a part of a lot of real world application. Therefore, I decided to write my own code as my own real world application. Due to my approach and use of variables, I developed a precise theoretical solution, yet the implemented code is limited to a double precision. In my case, this inaccuracy is insignificant, because I did not produce a picture with an extremely high resolution.

The resulting images look with in their limitations quiet realistic. However, the first thing I noticed was that no object reflects the entire incident light, for which reason the sphere did not appear realistic.
In order to make a more realistic image, I tried to reflect only 80% of the incident light. Therefore, I mixed the colors of the reflected rays with the color of the sphere 4:1.
Afterwards I want to know how realistic my pictures appear. Therefore, I asked 21 people to rate the realism in the four pictures bellow on a scale from 0 to 10 (higher is better).

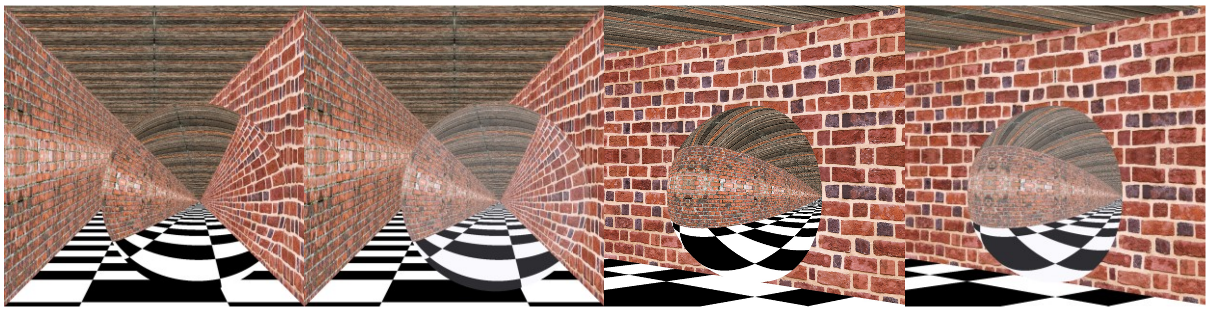


Figure 17 - Rendered images, with or without angle and 100% or 80% reflection

The data of the survey on photorealism of the four pictures can be found in the Appendix. In order to evaluate the data, the minimum, maximum, average and the standard deviation have been calculated from the sample data.

	Image without angle (100% reflection)	Image without angle (80% reflection)	Image with angle (100% reflection)	Image with angle (80% reflection)
Minimum	2	2	1	1
Maximum	8	8	9	8
Average	4.52	5.19	4.61	4.95
Standard deviation	1.83	1.99	2.38	2.27

From the knowledge of the different average, we can deduce that the images are in the middle between photorealistic and not at all realistic. Furthermore, the image with 100% reflection is less realistic. Nevertheless, due to the wide gap between the minimum and maximum value, we can state that the opinions on the photorealism of the images differ. Additional we can deduce from the standard deviations that the view on images with an angle are more discussed than the others.
As a conclusion we can say that the results are partly realistic, beyond that the spheres with only 80% reflection look more realistic than the others. Therefore, I managed it to render a partly photorealistic image with the use of vectors.
Since the image is not perfectly photorealistic, I can trace this back to the limitations I had to make. These limitations reduce the photorealism of the resulting image, such as the lighting or the neglect of shadows. So I can improve the photorealism by adding these concepts.

After all this is only the tip of the iceberg, modern ray tracing contains a lot more. For example, modern ray tracing contains concepts like shading models, antialiasing, indirect illumination and different object as a part of the scene.[23]

Apart from the photorealism of the picture, I noticed the high rendering times. Therefore, it is necessary to mention, that the applications of ray tracing are more suitable for non-real-time application, since ray tracing involves a lot of calculation. Additionally I started guessing why it took so long to calculate this image.
One of my main idea was that I have written a code, which has been performed serial (on step after the other). Therefore, I could not take advantage of the multithreading capabilities of my CPU, like other ray tracing applications. It also could have been better to process the calculations on the GPU ("Graphic processing unit") instead of the CPU ("Central processing unit").
The GPU has a huge advantage in multithreaded calculation (parallelized tasks) over the CPU, due to its specification for graphics calculations. This is due to the fact, that GPUs consist of a high amount of cores, for example the GTX 760 by NVIDIA has 1152 CUDA Cores with a boos clock of 1033 MHz, whether CPUs have only a small amount of Cores with a high speed, for example the Intel Xeon E3 1230 v3 has 4 cores with a boost speed of 3700 MHz . So in my case it would be better choice to perform raytracing on my GPU, because I can calculate the color of 1152 pixels at the same time, which would accelerate the performance by $\frac{1152}{\frac{3700Mhz}{1033Mhz}} = 321.63$ times. In contrast, to the multithreaded capabilities of my CPU that can accelerate the performance by 4 times.
Therefore, the image on the cover sheet would take a GTX 760 about $\frac{992min\ 5sec}{321.63\ pixel/sec} = 3min\ 5sec$.
Due to the comparison with CUDA cores instead of other units of the GPU, this calculation is suitable, since CUDA cores could be used for every application.[24]

Apart from the multithreaded optimization, it could be reasonable to group the object of the scene and decide whether the ray goes in the area of one group or another.[25] From this we could reduce the calculations. And obviously I could speed up the process be reducing the amount of pixels in one picture.

During the whole process, I learn a lot about the application of mathematics and concepts of computer science. Moreover, I noticed the deep connection between mathematics and computer sciences. Apart from this, I noticed other connections to different fields, such as art and product design. For example during my research I noticed the deep impact of ray tracing on the pop culture, due to the importance in the gaming or movie industry. Over all I have been impressed by the impact of ray tracing on different areas.

To conclude I have managed it to describe the basic concept of ray tracing with vectors. Furthermore, I have not only extended the basic concept with advanced functionalities like reflection and texture mapping, but also transferred the mathematical considerations into java. As a result I rendered a photorealistic image of a three dimensional scene with certain limitations.

[23] Interactive Ray Tracing with CUDA (NVIDIA)
[24] Ibidem
[25] Ibidem

9 SOURCES

9.1 BIBLIOGRAPHY:

Dr. K. A. Tsokos: Physics for IB Diploma, Cambridge 2010
Cambridge University Press, Cambridge, 5th edition
(English)

9.2 ONLINE BIBLIOGRAPHY:

Online Source: Executive summary about the animation and gaming industry in India (NASSCOM)
http://www.nasscom.in/sites/default/files/upload/63792/A_G_report_Executive_Summary.pdf , relevant request 21:42 3rd March 2015

Online Source: Moor's Law and what it means (Kiel University, Faculty of Engineering)
http://www.tf.uni-kiel.de/matwis/amat/semitech_en/kap_5/backbone/r5_3_1.html , relevant request 22:14 3rd March 2015

Online Source: Unreal Engine 4 (Epic Games, Inc.)
https://www.unrealengine.com/what-is-unreal-engine-4 , relevant request at 21:14 2nd March 2015

Online Source: Blenders powerful rendering engine Cycles (Blender Foundation)
http://www.blender.org/features/ , relevant request at 20:27 2nd March 2015

Online Source: RenderMan a high performance renderer by Pixar (Pixar)
http://renderman.pixar.com/view/p-renderman , relevant request at 20:12 2nd March 2015

Online Source: Advanced Ray Tracing – Computer Graphics 15-462 (Carnegie Mellon University, School of Computer Science)
http://www.cs.cmu.edu/afs/cs/academic/class/15462-s09/www/lec/13/lec13.pdf , relevant request at 18:12 20th February 2015

Online Source: Interactive Ray Tracing with CUDA (NVIDIA)
http://www.nvidia.com/content/nvision2008/tech_presentations/Game_Developer_Track/NVISION08-Interactive_Ray_Tracing.pdf , relevant request at 20:51 5th March 2015

Online Source: What is a pixel (WhatIs.com, TechTarget)
http://whatis.techtarget.com/definition/pixel , relevant request at 23:21 5th March 2015

Online Source: Unity 3D manual, textures and videos (Unity Technologies)
http://docs.unity3d.com/Manual/Textures.html , relevant request at 22:17 3rd March 2015

Online Source: Hidden Surface Algorithms – CSE 457 (University of Washington, Computer Science & Engineering)
http://courses.cs.washington.edu/courses/cse457/06sp/lectures/hidden-surfaces.pdf , relevant request at 12:21 21th February 2015

Online Source: Corpuscular theory of light (Wikipedia)
http://en.wikipedia.org/wiki/Corpuscular_theory_of_light , relevant request at 21:28 20th February 2015
The references of Wikipedia article such as http://plato.stanford.edu/entries/newton/ have been checked to verify its content.

Online Source: Dot Product (Wolfram MathWorld)
http://mathworld.wolfram.com/DotProduct.html , relevant request at 13:19 21th February 2015

Online Source: Cross Product (Wolfram MathWorld)
http://mathworld.wolfram.com/CrossProduct.html , relevant request 12:51 21th February 2015

Online Source: Implicit Differentiation (Massachusetts Institute of Technology)
http://web.mit.edu/wwmath/calculus/differentiation/implicit.html , relevant request 10:31 21th February 2015

Online Source: Java 8 documentation (Oracle)
http://docs.oracle.com/javase/8/ , relevant request 19:02 21th February 2015

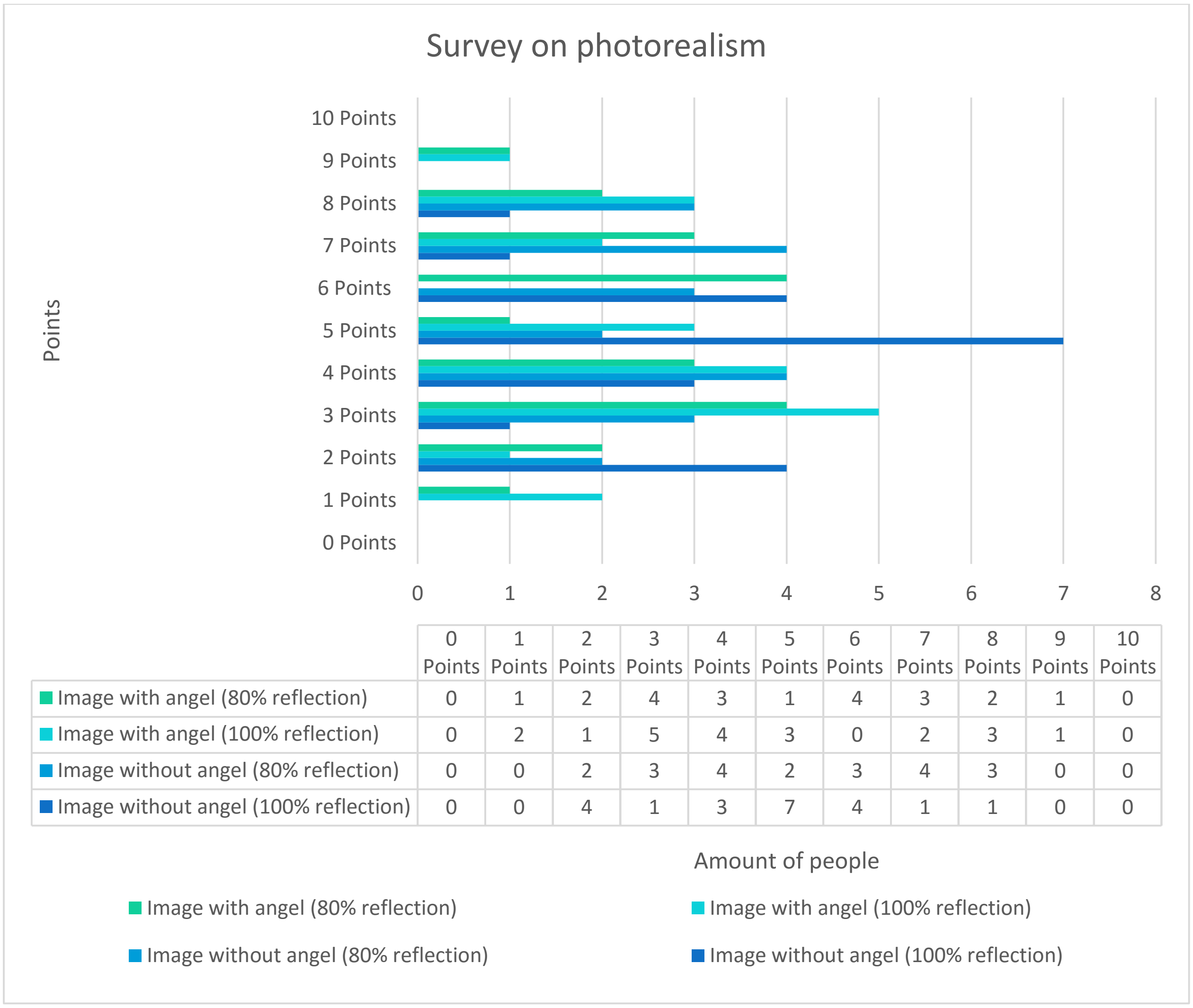
Online Source: Gauss elimination implementation (Sebastian Peuser)
http://wiki.freitagsrunde.org/Javakurs/%C3%9Cbungsaufgaben/Gau%C3%9F-Algorithmus/Musterloesung , relevant request 15:40 21th February 2015

Online Source: Instructional Ray-Racing Renderer – only Vector class has been used (Leonard McMillan, MIT 6.837 Fall -98)
http://groups.csail.mit.edu/graphics/classes/6.837/F98/Lecture20/RayTrace.java , relevant request 23:27 21th February 2015

9.3 LIST OF FIGURES

I have drawn all figures.

10 APPENDIX



	0 Points	1 Points	2 Points	3 Points	4 Points	5 Points	6 Points	7 Points	8 Points	9 Points	10 Points
Image with angel (80% reflection)	0	1	2	4	3	1	4	3	2	1	0
Image with angel (100% reflection)	0	2	1	5	4	3	0	2	3	1	0
Image without angel (80% reflection)	0	0	2	3	4	2	3	4	3	0	0
Image without angel (100% reflection)	0	0	4	1	3	7	4	1	1	0	0

Figure 17 - Survey on photrealism

Used Textures:

http://freestocktextures.com/texture/id/760

http://freestocktextures.com/texture/id/154

http://freestocktextures.com/texture/id/155

The full java code is published on GitHub (https://github.com/Langhalsdino/RayTracing).