Programming a Ray Tracer

Matura Paper
Lucas Edward Dodgson
Born 1999

Supervisor: Thilo Schlichenmaier
Co-supervisor: Thomas Müller

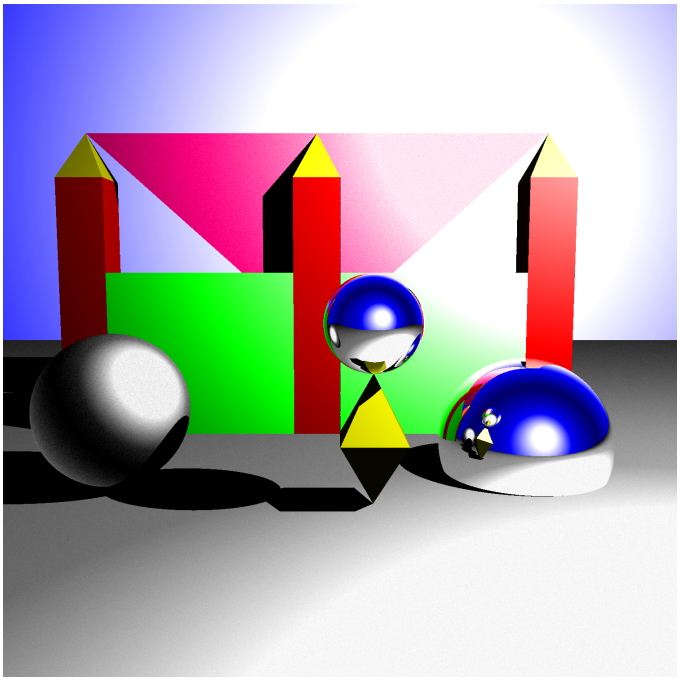


Figure 1: An image rendered with the ray tracer I programmed.

Freies Gymnasium Zürich,
Class of O61
October 2017

Table of contents

1 Foreword

My first experience with generating my own computer images was in 2015. This was in my applied mathematics class, where we wrote a python program that can draw basic images in two different perspectives.

Figure 2: A scene rendered with this method.

These images would only consist of lines and points and looked rather basic. But they were enough to make me realize that displaying computer graphics which look realistic is a lot more complicated than I had previously thought.

Figure 3: A computer generated caustic

Then in another project in 2016, we wrote a program to display caustics, an optical phenomenon caused by light rays that are refracted and reflected within a round object such as a drinking glass. This project further captured my interest in computer graphics and as someone who plays computer games I struggled to understand how realistic looking images could be generated in a fraction of a second.
Because of this I decided to pick the development of a ray tracing algorithm, a modern method of displaying graphics, as the topic for my matura paper, with the aim of further enriching my knowledge in this field.

At this stage I would like to express my gratitude to my supervisor Thilo Schlichenmaier for the support and constant advice he provided. I would also like to thank my parents for proofreading my paper and providing me with feedback.

2 Introduction

We live in a world in which more and more relies on computers. They control many aspects of our daily lives and with the rising demand for computers we search further and further for ways in which we can improve them.
One of the many areas in which we are constantly searching for improvements is computer graphics. For animated movies but also for the generation of single images and computer games we would like to display and generate images that are as realistic as possible. One common and modern method for doing this is ray tracing.
A ray tracer is a program that follows light rays to generate an image of a scene. It does this by reverse tracking the rays of light. It sends out rays from the camera/eye through pixels of the screen (this screen is the image generated in the end) and sees where they go. This is done because tracking the rays directly from the light source would mean a lot of rays end up not being relevant as they would not pass through the screen to the eye. Thus, by reverse tracking the rays, you can save a lot of computational power and increase simplicity without losing any relevant information for the image.

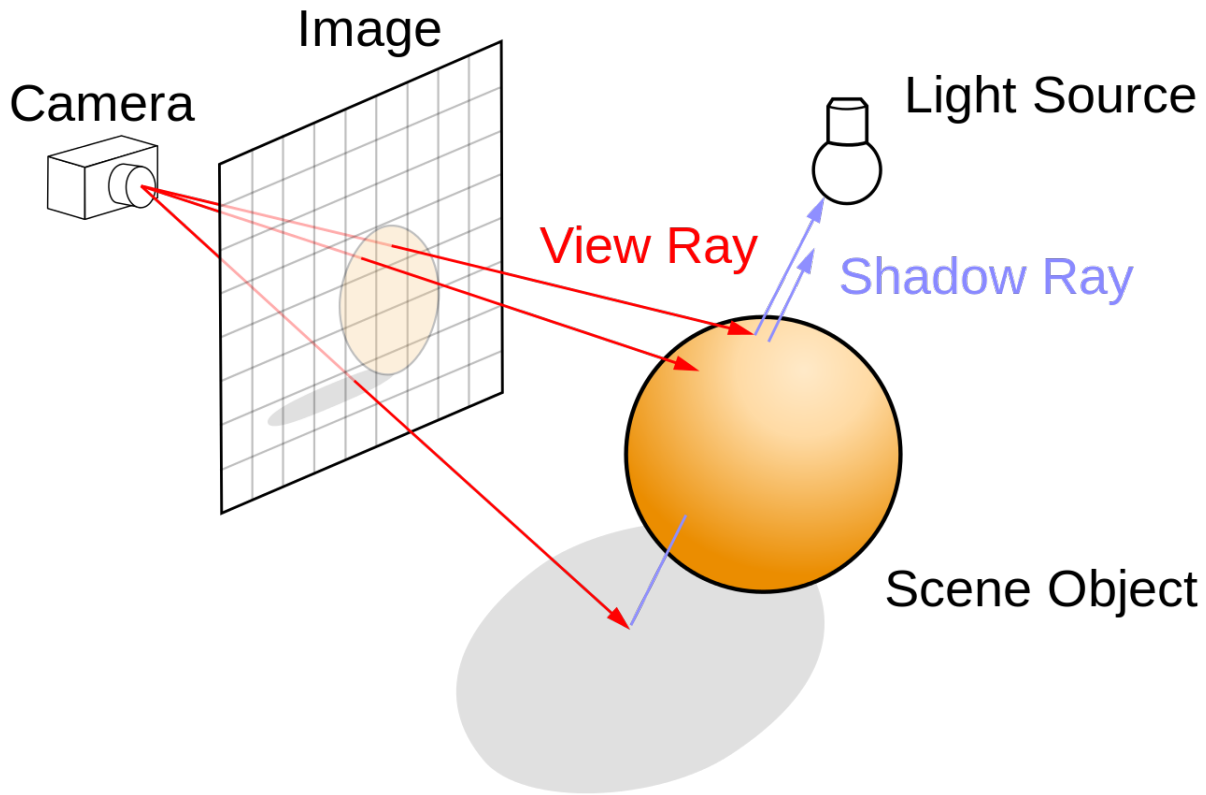


Figure 4: How a ray tracer works[1].

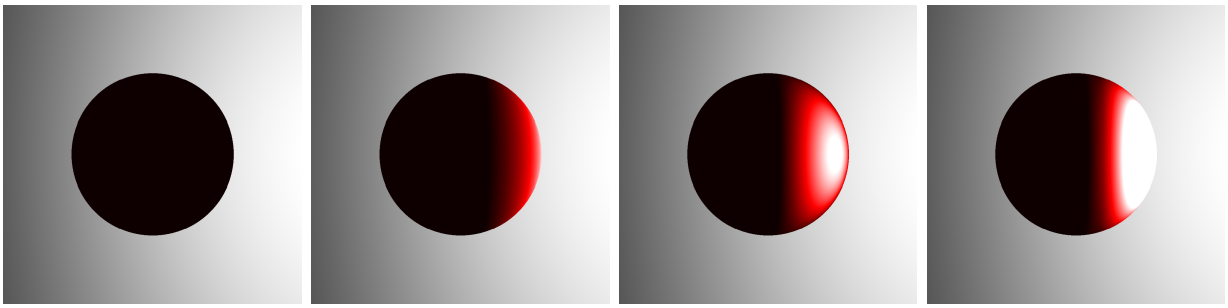
[1]Source: https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg

I decided that I was going to develop a ray tracing algorithm in the programming language `python` that would provide a toolbox with which complex images could be generated. In addition, I wanted to test the limit of my knowledge of python, mathematics, and to a certain extent physics, but also to uncover new information, gain a better understanding for computer graphics, and discover a real-world use for already learned phenomena.
I did this by creating various functions that act as the building blocks of the program. I then implemented objects, starting with spheres as calculating the point of intersection with them is relatively straightforward. I then added planes, triangles, and rectangles. Triangles are by far the most important object, as for commercial products like animated movies everything is built out of tiny triangles. To do this a very efficient algorithm is required and this presents an area in which more research is needed. While adding these new objects I also added new features to improve the program wherever possible, for example by trying to make the lighting look more realistic.
As there is a limit to what I could achieve in the given time, I decided I would limit the number of objects. There are also aspects that could be made more realistic with more time, for example I would have liked to add features like glass and water, that reflect and refract light.

3 Thoughts and ideas

3.1 Light intensity

To determine the brightness of a point in the scene, we need to look at the components of the light. Here these are ambient and diffuse lighting and the highlights. Each one of these numbers can be weighted differently depending on the material of the object.



Ambient Diffuse Highlights Combined

Figure 5: The different components of lighting.

The ambient brightness is a constant for every point in the scene. This means we assume every point will have at least some light reaching it that will be reflected into the eye. For points that are then affected by the diffuse lighting or the highlights, those factors will be added on top of the ambient brightness.

To determine the diffuse brightness of a point, we need to know the intensity of the light at that point. The intensity is dependent on the number of photons that reach an area, in other words the energy per area. Ideal diffuse lighting relies on the fact that the energy that is received is reflected in all directions evenly. Thus, the brightness does not depend on where the viewer is but just on the intensity of light at that point. If the light vector (shows direction of light ray) is equal to the normal vector multiplied by a constant c, meaning they show in the same or opposite direction (c can be negative), the area is minimal and the energy will be maximally concentrated. Thus, the intensity at this point will be big. If the light ray and the normal vector are at an angle of 85° to each other, the area for the same amount of energy will be a lot bigger than before. Thus, the intensity for any point in this area will be smaller and less light will be reflected in all directions. So, the light intensity for diffuse light becomes lower with a flatter impacting light ray.

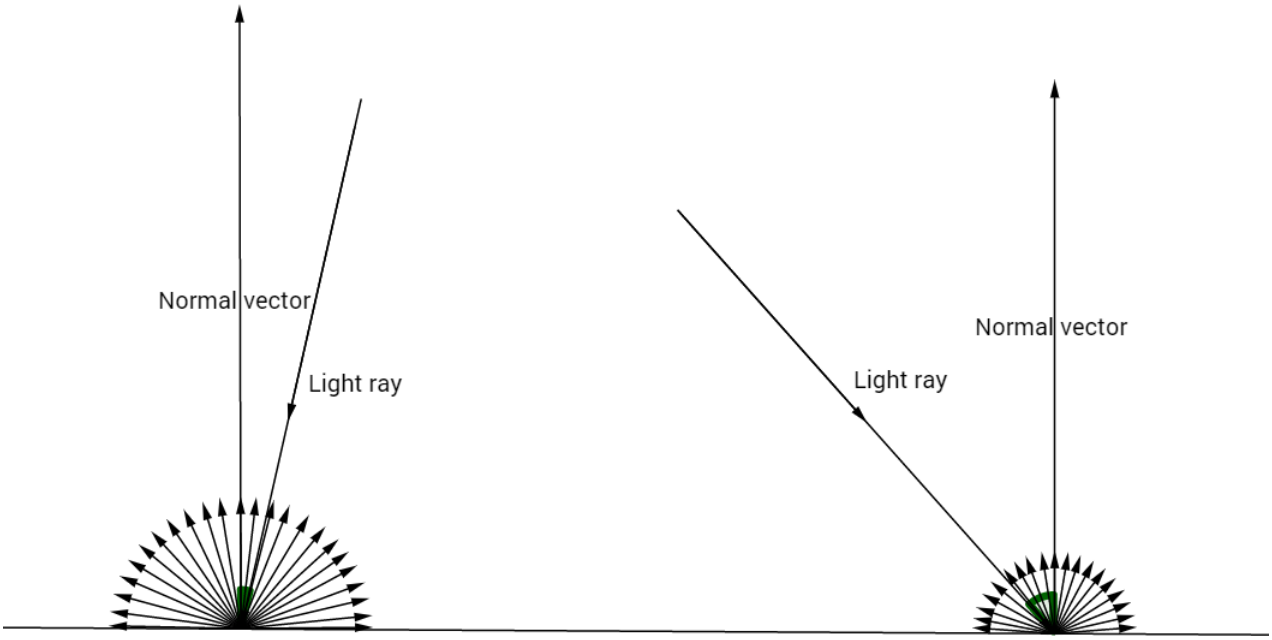


Figure 6: Comparison between two light rays and the diffuse lighting created by them.

To calculate the intensity of diffuse light at a point, one can use the cosine of the angle between the negative vector of the incoming light and the normal vector. Given a certain area, this area will grow the bigger the angle is.

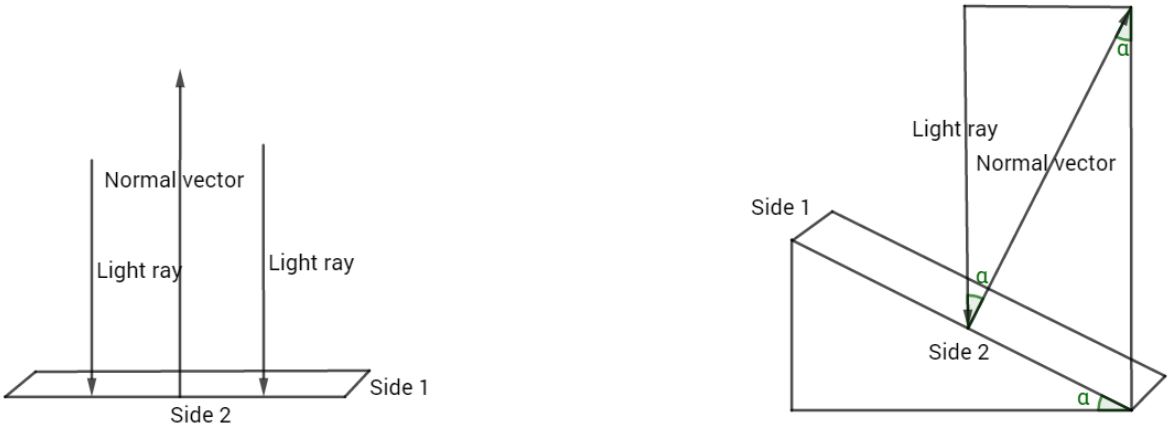


Figure 7: What happens to the sides of a rectangle if the angle grows.

So, if we assume the area of a rectangle is given by

$$A = s_1 \cdot s_2$$

Where s_1 and s_2 are the lengths of the sides. We assume that in the case where the angle between the negative normal vector and the incoming light is 0°, the two sides form a square. As the angle grows, one of the sides' length will stay the same (in this image Side

1), while the other will grow.

$$Side_2 = \frac{Side_1}{\cos a}$$

Thus, the intensity I can be given with the original area A and the amount of energy E:

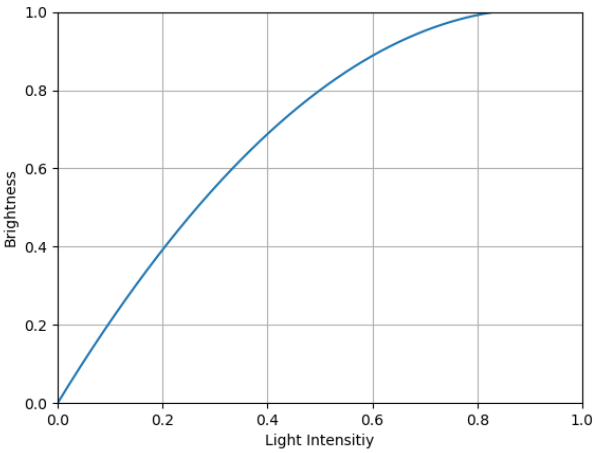
$$I = \frac{E}{A} \cdot \cos a$$

The angle a describes the slope of the plane, as well as the angle between the two vectors. And $\frac{E}{A} = 1$
This is known as Lamberts cosine law. Instead of using the cosine, you can also use the scalar product of the normalized vectors as this is equal to the cosine of the angle between them.

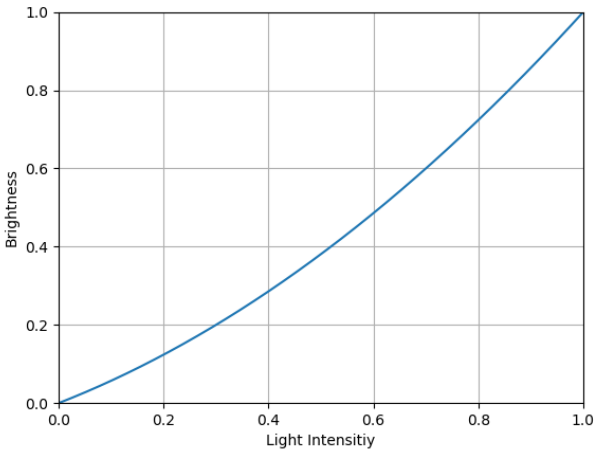
For the highlights you need to look at the reflected vector and the light ray vector. If these are facing the same direction you can set this part of the brightness to the maximum, as that means all the light that is reflected from an idealized surface will reach the eye. For the highlights, you can also use the scalar product of the two normalized vectors, this time of the reflected vector and the negative vector of the light ray. Again, if the angle is $0°$ (scalar product is 1), you get a case where the light is reflected and directly reaches your eye. Thus, the intensity would be maximal. If the angle then increases this would become lower and lower. Like this with the scalar product of the normalized vectors, you will have a good approximation for the intensity of the light that will in the end reach the eye.

3.2 Light function

Using the methods described above you can calculate the light intensity of a point. There is nothing wrong with setting the brightness to equal the intensity of the light at that point. But in certain situations it can look better if you instead look at a non-linear function for the brightness. This function should take a light intensity value between 0 and 1, and return a brightness between 0 and 1. In these examples $f(x)$ is the brightness of the point, and x is the intensity.



$$f(x) = -1.2 \cdot x^2 + 2.2 \cdot x$$



$$f(x) = 0.47 \cdot x^2 + 0.53 \cdot x$$

Figure 8: Two examples of light functions.

The first example shows a rapid growth at the start and then a plateau around a light intensity of 0.8 where the maximum brightness is reached. The second one shows a slower growth where a light intensity of 0.5 causes a brightness of 0.4

3.3 Mirrors

An ideal mirror will reflect 100% of the light that hits it. For this ray tracing algorithm, to turn an object into a mirror you need to treat it as a reflective object. For simplicity's sake we will treat mirrors as ideal. So, the light will continue on with the same brightness as before but now will be reflected along the plane of the mirror.
So, if you have a ray that is traced from the eye, this will be reflected by the mirror. Consequently, you can treat the reflected ray as the ray that needs to be traced to determine the colour of the pixel. You still draw the pixel at the same place with the resulting colour and brightness. Now you also need to be careful because a ray could theoretically become trapped between two mirrors and the program could keep on running forever. To avoid this, you just need to set a limit on the number of reflections.

4 Mathematical theory

4.1 Point of intersection between a sphere and a light ray

In this section we will determine the point of intersection between a sphere and a light ray. Let $\vec{m} = (x_0, y_0, z_0)^T$ be the centre of the sphere, the radius is r, and $\vec{p} = (x_3, y_3, z_3)^T$ be a point that lies on the sphere. The sphere can then be given by the equation:

$$(x_0 - x_3)^2 + (y_0 - y_3)^2 + (z_0 - z_3)^2 = r^2$$

This can be expanded, you then get:

$$x_3{}^2 - (2 \cdot x_3 \cdot x_0) + x_0{}^2 + y_3{}^2 - (2 \cdot y_3 \cdot y_0) + y_0{}^2 + z_3{}^2 - (2 \cdot z_3 \cdot z_0) + z_0{}^2 = r^2$$

The coordinates for the eye point are $\vec{a} = (x_1, y_1, z_1)^T$ and the direction of the ray is given as $\vec{v} = (x_2, y_2, z_2)^T$

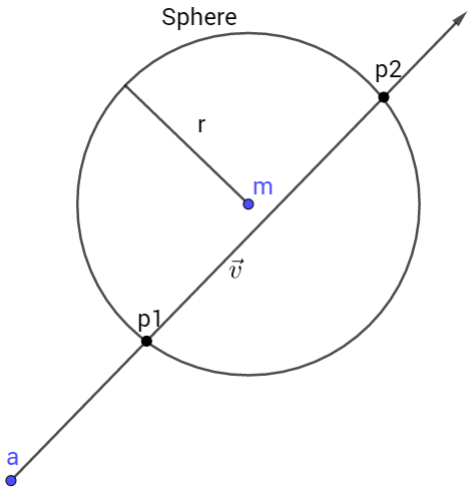


Figure 9: A ray intersecting a sphere.

As we are looking for the point $\vec{p}$ of intersection between $\vec{v}$ and the sphere we define:

$$x_3 = x_1 + t \cdot x_2$$
$$y_3 = y_1 + t \cdot y_2$$
$$z_3 = z_1 + t \cdot z_2$$

In the sphere equation x, y, and z have the same factors. Because of this the next steps

will just be looking at x, yet the same steps can be taken with y and z.

$$\begin{aligned}
&(x_1 + t \cdot x_2)^2 - 2 \cdot x_1 \cdot x_0 - 2 \cdot x_2 \cdot t \cdot x_0 + x_0^2 \\
&= x_1^2 + t^2 \cdot x_2^2 + 2 \cdot x_1 \cdot t \cdot x_2 - 2 \cdot x_1 \cdot x_0 - 2 \cdot x_2 \cdot t \cdot x_0 + x_0^2 \\
&= x_2^2 \cdot t^2 + t(2 \cdot x_1 \cdot x_2 - 2 \cdot x_2 \cdot x_0) + x_1^2 + x_0^2 - 2 \cdot x_1 \cdot x_0
\end{aligned}$$

If you do the same with y and z the overall equation now is:

$$\begin{aligned}
&t^2 \cdot (x_2^2 + y_2^2 + z_2^2) + t(2 \cdot x_1 \cdot x_2 - 2 \cdot x_2 \cdot x_0 + 2 \cdot y_1 \cdot y_2 - 2 \cdot y_2 \cdot y_0 + 2 \cdot z_1 \cdot z_2 - 2 \cdot z_2 \cdot z_0) \\
&+ (x_1^2 + x_0^2 - 2 \cdot x_1 \cdot x_0 + y_1^2 + y_0^2 - 2 \cdot y_1 \cdot y_0 + z_1^2 + z_0^2 - 2 \cdot z_1 \cdot z_0 - r^2) = 0
\end{aligned}$$

This is a quadratic equation that can be solved for t. In most cases you get two or no answers. To find the point of intersection between a ray and the sphere you will want the smaller positive t value. With this t the point of intersection can be calculated.

One could also calculate the point of intersection using only vector operations and not looking at the different components of the respective vectors.
A sphere with a radius r and the centre $\vec{m}$ is given. $\vec{p}$ is the point of intersection that is looked for. For any point on the sphere, the length $\vec{m} - \vec{p}$ will always be the same (r). Thus, the sphere is given by:

$$r^2 = (\vec{p} - \vec{m})^2$$

and

$$\vec{p} = \vec{a} + t \cdot \vec{v}$$

By combining and expanding these two equations you get:

$$\begin{aligned}
r^2 &= ((\vec{a} + t \cdot \vec{v}) - \vec{m})^2 \\
r^2 &= (\vec{a} + t \cdot \vec{v})^2 + \vec{m}^2 - 2(\vec{m} \cdot \vec{a} + \vec{m} \cdot t \cdot \vec{v}) \\
r^2 &= \vec{a}^2 + t^2 \cdot \vec{v}^2 + 2 \cdot \vec{a} \cdot \vec{v} \cdot t + \vec{m}^2 - 2 \cdot \vec{m} \cdot \vec{a} - 2 \cdot \vec{m} \cdot t \cdot \vec{v} \\
0 &= t^2 \cdot (\vec{v}^2) + t \cdot (2 \cdot \vec{a} \cdot \vec{v} - 2 \cdot \vec{m} \cdot \vec{v}) + (\vec{a}^2 + \vec{m}^2 - 2 \cdot \vec{m} \cdot \vec{a} - r^2)
\end{aligned}$$

This is the same quadratic equation as above, just given with the vectors and not their components.

4.2 Finding the reflected light ray

In this section, we calculate the reflected vector w, which is required to calculate the highlights, of a vector v that is reflected off a point that lies on an object. To do this we also need to know the normal vector n.

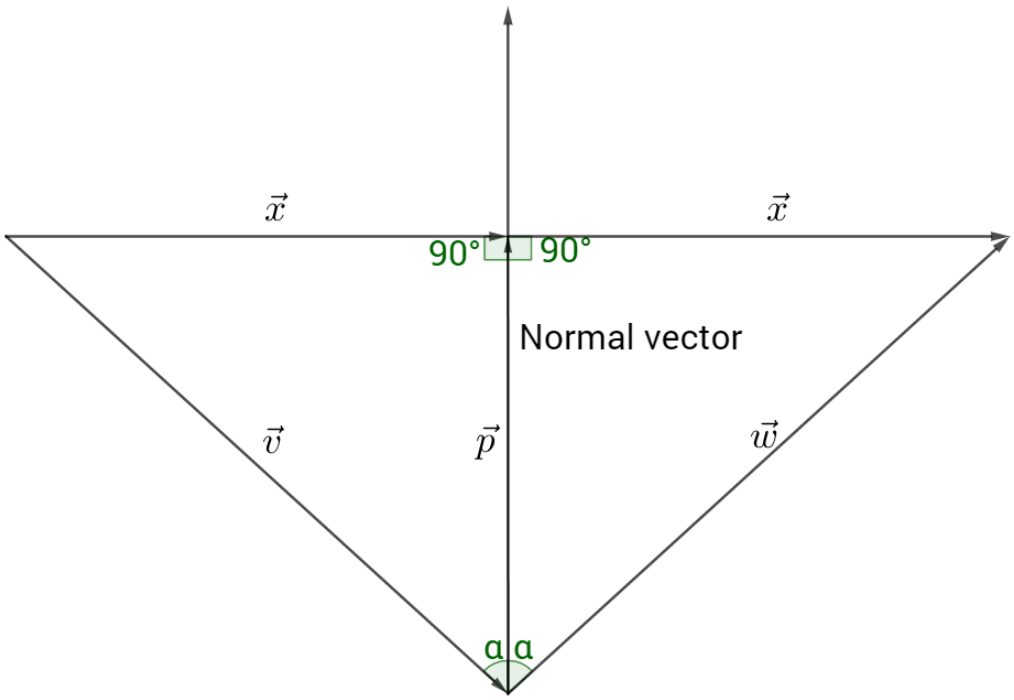


Figure 10: How $\vec{v}$ is reflected.

We introduce $\vec{x}$ as a help to do this. We know that:

$$\vec{x} \cdot 2 = \vec{v} + \vec{w}$$

and that:

$$\vec{x} = \vec{v} + t \cdot \vec{n}$$

for an unknown t.
So:

$$\vec{v} + \vec{w} = 2 \cdot (\vec{v} + t \cdot \vec{n})$$

We define:

$$\vec{p} = t \cdot \vec{n}$$

Which is the vector originating from the same point as $\vec{n}$ and leading to the intersection of $\vec{n}$ and $\vec{x}$.

$$
\begin{aligned}
0 &= \vec{p} \cdot \vec{x} \\
\vec{v} &= \vec{x} - \vec{p} \\
\vec{x} &= \vec{v} + \vec{p} \\
0 &= \vec{p} \cdot (\vec{v} + \vec{p}) \\
0 &= \vec{p} \cdot \vec{v} + \vec{p}^2 \\
0 &= t \cdot \vec{n} \cdot \vec{v} + t \cdot \vec{n} \cdot t \cdot \vec{n} \\
0 &= t \cdot (\vec{n} \cdot \vec{v} + t \cdot \vec{n}^2)
\end{aligned}
$$

One answer to this equation would be t = 0, but that would be the point of intersection from which the ray is reflected. This is not what we are looking for. Thus, we say $t \neq 0$

$$
\begin{aligned}
\vec{n} \cdot \vec{v} &= -\vec{n}^2 \cdot t \\
t &= -\frac{\vec{n} \cdot \vec{v}}{\vec{n}^2}
\end{aligned}
$$

So:

$$
\begin{aligned}
\vec{p} &= -\frac{\vec{n} \cdot \vec{v}}{\vec{n}^2} \cdot \vec{n} \\
t \cdot \vec{n} &= -\frac{\vec{n} \cdot \vec{v}}{\vec{n}^2} \cdot \vec{n}
\end{aligned}
$$

Thus:

$$
\vec{w} = \vec{v} + 2 \cdot (-\frac{\vec{n} \cdot \vec{v}}{\vec{n}^2}) \cdot \vec{n}
$$

Since we know $\vec{v}$ and $\vec{n}$ it is now possible to calculate the reflected vector $\vec{w}$.

4.3 The placement of the screen

Instead of setting the screen in a static location, we want to move the eye and the screen with it. This allows us to observe a scene from different angles.
For this we define a centre point $\vec{c}$. This point dictates in what direction the eye is looking. We then place the scene on the plane between $\vec{c}$ and $\vec{a}$ which has $\vec{c}-\vec{a}$ as its normal vector. This screen is now a two-dimensional plane in three-dimensional space, thus we need to define a special coordinate system, in this case we use the two vectors $\vec{u}$ and $\vec{v}$.
We want to define $\vec{u}$ so that $\vec{u} \cdot \vec{n} = 0$, and $\vec{v} = \vec{u} \times \vec{n}$.

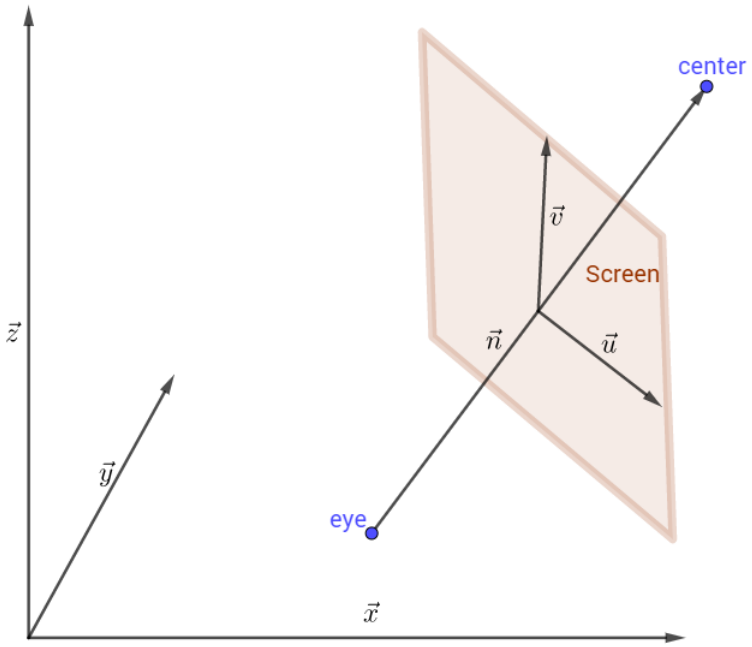


Figure 11: The new $\vec{u}/\vec{v}$ coordinate system.

With $\vec{n} = (c_x - a_x, c_y - a_y, c_z - a_z)^T$ we define $\vec{u} = (u_x, u_y, 0)^T$ and $\vec{v} = (v_x, v_y, v_z)^T$. We want to solve

$$u_x \cdot n_x = -u_y \cdot n_y$$

for u_x. We need to make sure $n_x \neq 0$. If $n_x = 0$ we want to set $u_y = 0$ and $u_x = 1$. Otherwise, we just set $u_y = 1$ and then calculate u_x. Once we have $\vec{u}$ we can calculate

$$\vec{v} = \vec{u} \times \vec{n}$$

So:

$$\begin{aligned} v_x &= u_y \cdot n_z \\ v_y &= -u_x \cdot n_z \\ v_z &= u_x \cdot n_y - u_y \cdot n_x \end{aligned}$$

For the program, we need to know the bottom left point of our screen. To do this, we first calculate the middle point of our screen,

$$\vec{p} = \vec{a} + \frac{\vec{n}}{2}$$

To find the bottom corner of the screen, we use normalized $\vec{u}$ and $\vec{v}$.

$$\vec{lp} = \vec{p} - \frac{height}{2} \cdot \vec{v} - \frac{width}{2} \cdot \vec{u}$$

Height and width are two numbers we chose to define the size of the screen.

4.4 Point of intersection between a triangle and a light ray

In this section we will look at how you can check if a ray and a triangle intersect, and how to calculate this point.
Calculating the point of intersection between a triangle and a ray requires two main steps. You first need to calculate the point of intersection between the ray and the plane in which the triangle lies. Then you need to have a method to check if this point lies inside or outside the triangle.
You have a triangle with three points.
Let $\vec{a} = (a_x, a_y, a_z)^T$, $\vec{b} = (b_x, b_y, b_z)^T$, and $\vec{c} = (c_x, c_y, c_z)^T$
Note: in this section we will use: $e\vec{y}e = (a.x, a.y, a.z)^T$.
The normal vector for the plane will be the cross product of $\vec{b} - \vec{a}$ and $\vec{c} - \vec{a}$

$$\vec{n} = (n_x, n_y, n_z)^T = (\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})$$

The plane is then given by:

$$E: \ 0 = \vec{n} \cdot \vec{p} - \vec{n} \cdot \vec{b}$$

Now you insert the point

$$\vec{p} = (a.x + t \cdot v_x, a.y + t \cdot v_y, a.z + t \cdot v_z)^T$$

We define $d = \vec{n} \cdot \vec{b}$

$$\begin{aligned} E: \ & 0 = \vec{n} \cdot (a.x + t \cdot v_x, a.y + t \cdot v_y, a.z + t \cdot v_z)^T - d \\ & 0 = n_x \cdot t \cdot v_x + n_x \cdot a.x + n_y \cdot t \cdot v_y + n_y \cdot a.y + n_z \cdot t \cdot v_z + n_z \cdot a.z - d \\ & d - \vec{a} \cdot \vec{n} = t \cdot (\vec{n} \cdot \vec{v}) \end{aligned}$$

Now if the factor by which t is multiplied is 0, then the ray lies inside the plane (scalar product of vector and normal = 0). In that rare case, you can say there is no intersection point. Then you continue with:

$$t = \frac{d - \vec{a} \cdot \vec{n}}{\vec{n} \cdot \vec{v}}$$

That was the first part, using t you can now easily calculate the intersection point.
Once you have that point, to check whether the point is inside the triangle you can compare the areas of the triangles, since the following condition is only true if the point lies within the triangle $\triangle ABC$.

$$\triangle ABC = \triangle PAB + \triangle PBC + \triangle PAC$$

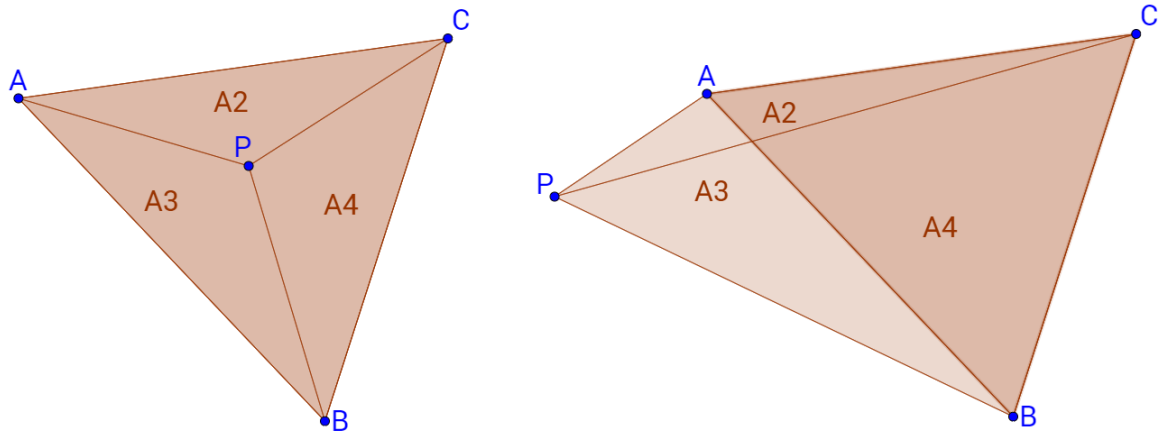


Figure 12: Comparison of areas between when the point P lies in the inside (left) or outside (right) of the triangle ABC.

To calculate these areas, Herons formula can be used. Herons formula is a way of calculating the area of a triangle,

$$A = \sqrt{s \cdot (s-a) \cdot (s-b) \cdot (s-c)}$$

where s is half the circumference.

$$s = 0.5 \cdot (a+b+c)$$

4.5 Checking if a point lies inside or outside a polygon

The method of checking if a point lies inside or outside a triangle by comparing the area can be used for other shapes as well. But there is a more efficient way to do this that works for any polygon:
Once you have the point on the plane in which the triangle lies, you want to determine the vectors $\vec{AB}, \vec{AP}, \vec{BC}, \vec{BP}, \vec{CA}, \vec{CP}$ and then the cross products $\vec{AB} \times \vec{AP}, \vec{BC} \times \vec{BP}$, $\vec{CA} \times \vec{CP}$. Now as long as $\vec{P}$ is inside the triangle, the vectors $\vec{AP}, \vec{BP}, \vec{CP}$ will always be either to the left or right of $\vec{AB}, \vec{BC}, \vec{CA}$ respectively. Because of that, all their cross products will be looking in the same direction, which is also the direction the normal vector is pointing in. Thus, the cross products, if the point lies inside, will be pointing in the same overall direction as the normal vector. The difference will always be smaller than 90 degrees. Because of that the scalar product of these vectors to the normal vector (and also these vectors to each other) will be bigger than 0 if P lies inside the triangle. If P lies outside the triangle at least one of the cross products will be showing in the opposite direction.

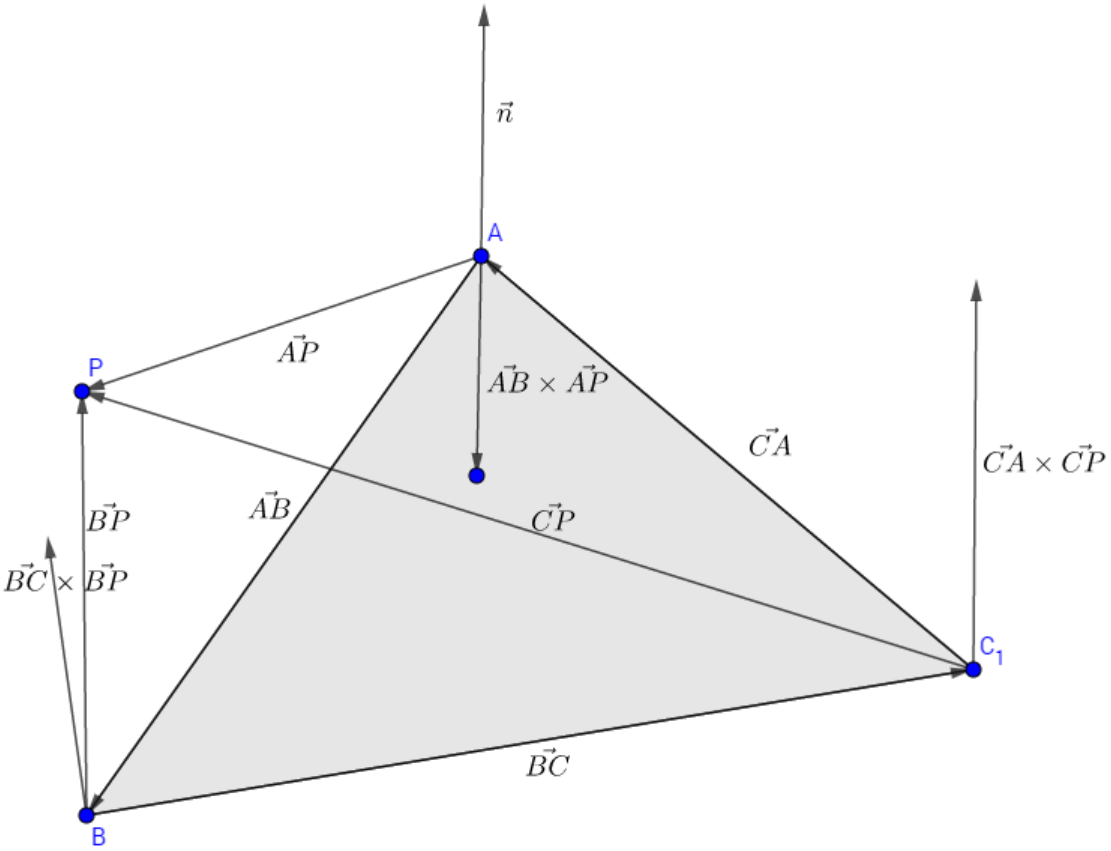


Figure 13: If the point lies outside the polygon, in this example $\vec{AB} \times \vec{AP}$ will point in the opposite direction as the others, thus the scalar product will be smaller than 0.

5 Stages of development of the python code

5.1 Vector and sphere classes

The following code creates a function that solves quadratic equations and the python classes `Vector` and `Sphere`.
The quadratic equation function first checks how many real answers there are to the equation, 0, 1 or 2. If there are 0 we can just return $(-1, -1)$ as our answers because we are only looking for positive answers we will filter out the negative ones after calling the function. Otherwise it calculates all the answers.
In the vector class you have functions to add and subtract two vectors with each other and to multiply a vector by a factor t. You then have the possibility of returning the components of a vector and the scalar product of two vectors as well as the cross product of two vectors. You also have the possibility to normalize a vector.
In the sphere class, you have a function that calculates the value `t` with which you multiply the vector `v` from the point `a` to reach the intersection point as outlined on page 9. You then have a function that returns the coordinates of this intersection point and one to return the normal vector for the tangential plane at this point.
This is the current code:

```
from math import *
def quadr_equation (a,b,c):
        "Returns the real answers for the equation ax^2+bx+c = 0"
        d = b**2-4*a*c
        if  d<0:
                return (-1,-1)
        elif  d == 0:
                return [-b/(2*a),-1]
        else:
                x1 = (-b+ sqrt(d))/(2*a)
                x2 = (-b- sqrt(d))/(2*a)
        return  [x1, x2]

class Vector:
        def __init__(self,x,y,z):
                self.x=x
                self.y=y
                self.z=z

        def __add__(self, other):
                """Additions operator + is overloaded."""
                return Vector(self.x+other.x,self.y+other.y,
                  self.z+ other.z)
```

```
def __sub__(self, other):
        """Subtractions operator - is overloaded."""
        return Vector(self.x-other.x,self.y-other.y,
          self.z-other.z)

def __mul__(t,self):
        """Multiplications operator * is overloaded."""
        return Vector(t*self.x,t*self.y,t*self.z)

def components(self):
        """Returns a list of the components of the vector."""
        return [self.x, self.y, self.z]

def scalarp(self, other):
        "Scalar product"
        return self.x*other.x+self.y*other.y+self.z*other.z

def cross(self,other):
        "cross product"
        xcomp = self.y*other.z-self.z * other.y
        ycomp = self.z*other.x-self.x *other.z
        zcomp = self.x*other.y-self.y*other.x
        return Vector(xcomp,ycomp,zcomp)

def length(self):
        "returns the length of the vector"
        return float((sqrt(self.scalarp(self))))

def normalize(self):
        "Sets the vector to the length 1"
        r= self.length()
        return(self*(1/r))
```

```
class Sphere:
        def __init__(self,m,r,colour):
                """Creates a Sphere with the centre m (a vector),
                  radius r and a colour"""
                self.m=m
                self.r=r
                self.colour= colour
        def  Intersection(self,a,v):
                "A = eyepoint, v = light vector"
                m = self.m
                Qa = v.scalarp(v)
                b = 2*(a.scalarp(v)-m.scalarp(v))
                c = a.scalarp(a)+m.scalarp(self.m)-2*m.scalarp(a)-
                  self.r**2
                t= quadr_equation(Qa,b,c)
                #The following returns the smaller, positive t value
                t1 = t[0]
                t2 = t[1]
                if t1 < 0:
                        if t2 < 0:
                                return None
                        else:
                                return t2
                else:
                        if t2 < 0:
                                return t1
                        elif t1 < t2:
                                return t1
                        else:
                                return t2
        def IntersectionP(self,a,v,t):
                "A= eyepoint, v= light vector, t= factor (calculate with
                  Intersection function)"
                nV = v*t
                return(a+nV)
        def Normalvector(self,s):
                "s is the point of intersection"
                return (s-self.m)
```

5.2 First image

The following code, allows the displaying of a sphere. This, at the current stage looks like a circle since the colour and its brightness is constant for every point on the sphere. To display the sphere the canvas library was used. This library allows the setting up of a screen. On this you can then add various objects, although only rectangles were used here.

```
def Pixel(p,col):
        """Creates a pixel with the centre p and colour col."""
        x,y=p[0],p[1]
        s,t=s0+x,t0-y
        return cv.create_rectangle(s-0.5*1, t-0.5*1, s+0.5*1,
          t+0.5*1, fill=col, outline=col)

#Eye coordinates
a = Vector(40,-300,40)

#The sphere
s = Sphere(Vector(40,50,40),40,"red")

#This sets the resolution of the image
hf=Tk()
width = 600
height = 600
cv=Canvas(hf, width=width, height=height)
cv.pack()
#This sets the coordinates for the lowest and the highest point of the
   screen
screenlow = Vector(a.x-0.5*width, 0, a.z-0.5*height)
screenhigh = Vector(a.x+0.5*width,0,a.z+0.5*height)
#This lets us use a more standard coordinate system, not the default one
    included in the canvas library
s0,t0=int(1/2*width-a.x),int(a.z+1/2*height)
```

Right now, the screen lies on the y axis, thus the y coordinates of all the pixels are 0.
Point A here would be the `screenlow` coordinate, point E_3 would be the `screenhigh`. The following code then loops over all the pixels, starting with the bottom row, in the scene and sends a ray through each one. It then checks if the ray intersects with the sphere or not. If not it draws the pixel black, if it does the pixel is drawn in the colour of the sphere.

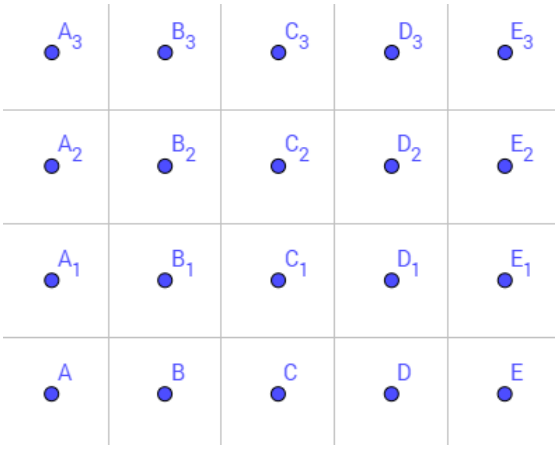


Figure 14: A five by four pixel screen.

```
for i in range(int(screenlow.x),int(screenhigh.x)):
        j = screenlow.z
        for j in range(int(screenlow.z),int(screenhigh.z)):
                v = Vector(i-a.x,-a.y,j-a.z)
                t = s.Intersection(a,v)
                if t == None:
                        Pixel([i,j],"black")
                else:
                        Pixel([i,j],s.colour)
```

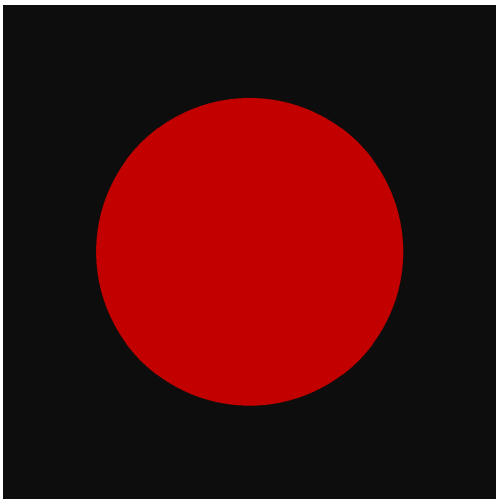


Figure 15: A first image.

5.3 Brightness functions

Two new functions `reflectedray` and `tolight` are defined as part of the `sphere` class. `reflectedray` calculates the reflected vector with the help of the normal vector and the light vector (as outlined on page 11). The second function returns the vector that goes from the intersection point to the light source, which will be used to calculate the brightness.

```
        def reflectedray(self,v,n):
                a = v.scalarp(n)
                b = n.scalarp(n)
                c = a/b
                p = n*(2*c)
                w = v-(p)
                return w
        def tolight(self,intersection,light):
                return light-intersection
```

The following functions are used to decide the brightness of every pixel. The function `highlights` takes the normalized versions of the reflected vector and the vector leading to the light and returns the highlight component of the lighting which will be a number between 0 and 1. The function `diffuse` takes the normalized versions of the normal vector and the vector leading to the light. It then returns that component of the lighting. The function `defbrightness` takes a brightness parameter, the normal vector for the tangential plane, the normalized vector leading to the light, the reflected vector, and the constant for ambient light. It then adds together the highlights multiplied by the brightness parameter of the object, the diffuse brightness multiplied by the brightness parameter of the object, and the ambient brightness into one. Finally, `backbrightness` calculates a brightness for the background. The background facing the direction of the light source will be brighter than elsewhere. This again takes two normalized vectors, the vector leading directly from the eye to the light source and the vector containing the direction of the light ray.

```
def highlights(refvector, lightvec):
        "The reflected light vector and the vector leading directly to
          the intersection point"
        r = refvector
        l = lightvec
        z = r.scalarp(l)
        if z < 0:
                return 0
        else:
                return z
```

```
def diffuse(normalvector,lightvec):
        "Normal vector for the plane, and the vector leading
          from the intersection point to the light."
        n = normalvector
        l = lightvec
        z = n.scalarp(l)
        if z < 0:
                return 0
        else:
                return z
#c can be used later to determine the brightness  of an object. Ambient
   is a global constant.
def defbrightness(c, n, l, r,ambient):
        "Constant for the object, the normal vector for the plane,
          vector leading to the light, the reflected vector,
          the ambient light."
        h = highlights(r,l)
        d = diffuse(n,l)
        bn = (d+h) * c
        return min(bn+ambient,1)
def backbrightness(l,r,ambient):
        "vector leading to the light, vector going through the pixel,
          the ambient light"
        h = r.scalarp(l)
        if h < 0:
                return ambient
        elif h + ambient > 1 :
                return 1
        else:
               return h + ambient
```

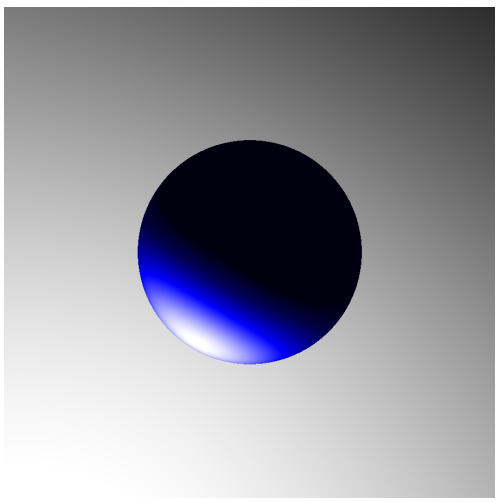


Figure 16: A blue sphere with all the components of lighting.

5.4 Multiple objects

The following function allows the rendering of as many objects as the user likes. It loops over a list of objects and first calculates the `t` value for the point of intersection. This is either `none` or a number. It then looks for the smallest `t` value of all the objects and either returns the object and its `t` value or `none` in the case this ray intersects no object in the scene.

```
def Objectsort(a,v,objects):
        "Takes the light ray and checks which object, if any, it touches
           first. a = eye vector, v = direction of ray,
          objects = list of all the objects."
        tlist = []
        for i in objects:
                t = i.Intersection(a,v)
                tlist.append(t)
        loc = None
        vt = math.inf
        leng = len(tlist)
        for i in range(leng):
                value = tlist[i]
                if value != None:
                        if value < vt:
                                vt = value
                                loc= i
        if loc == None:
                return None,None
        else:
                return objects[loc],vt
```

The code that loops over all the pixels has been updated so that it now has a `if` condition that prints the percentage completed every 10 rows. This means that in large images with lots of objects you can get a feel of how long the remaining time is.

```
per = 0
leng = 100/width
count = 0
for i in range(int(screenlow.x),int(screenhigh.x)):
        if count == 10:
                print(per)
                count = 0
        count += 1
        per += leng
        for j in range(int(screenlow.z),int(screenhigh.z)):
                v = Vector(i-a.x,-a.y,j-a.z)
```

```
o,t= Objectsort(a,v,Objects)
if o == None:
        Colour = Color("black")
        v = v.normalize()
        v2 = lightsource-a
        v2= v2.normalize()
        brightness = backbrightness(c,v,v2,ambient )
        Colour.luminance = brightness
        Pixel([i,j],Colour)
else:
        y = o.IntersectionP(a,v,t)
        n = o.Normalvector(y)
        n = n.normalize()
        r = o.reflectedray(v,n)
        r = r.normalize()
        c= o.constant
        light = o.tolight(y,lightsource)
        light = light.normalize()
        Colour = Color(o.colour)
        brightness = defbrightness(c,n,light,r,ambient)
        Colour.luminance = brightness
        Pixel([i,j],Colour)
```

The following image contains multiple spheres, including a big pink one that is placed around the whole scene.

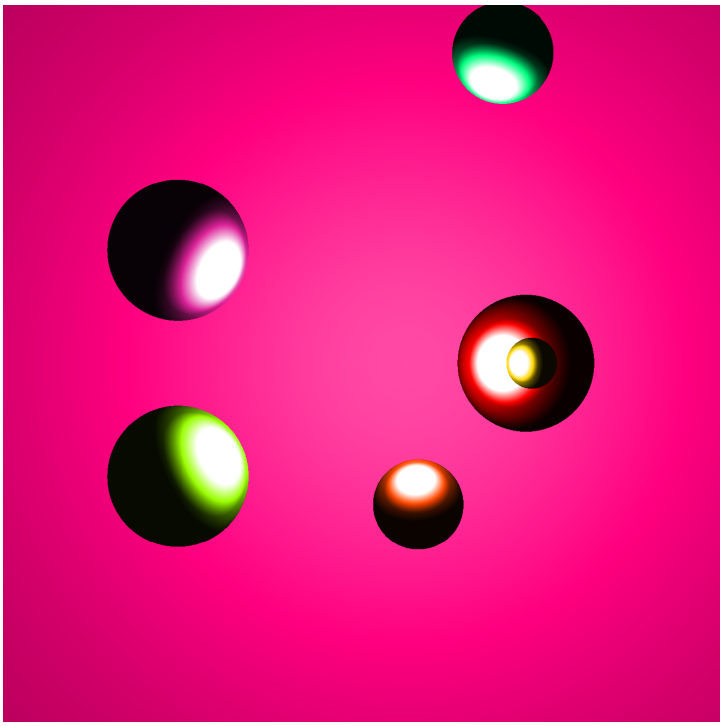


Figure 17: A scene consisting of spheres.

5.5 Shadows

To include shadows, you can check if a certain point has a clear line of sight to the light source or if there is another object in the way. If that is the case no light rays would reach this object and you would only want to include the ambient brightness and not the other two components.

```
def isshadow(tolight, intersectionp, Objects,obj):
        "Checks if a point of an object lies in a shadow or not,
          warning tolight cannot be normalized or this will be wrong!"
        z = list(Objects)
        z.remove(obj)
        o2,t = Objectsort(intersectionp,tolight,z)
        if t == None:
                return False
        elif t >= 1:
                return False
        else:
                return True
```

Now you just need to check if this returns true or false in the brightness function.

```
if n == True:
        brightness = ambient
else:
        light = light.normalize()
        n = (o.Normalvector(y)).normalize()
        r = (o.reflectedray(v,n)).normalize()
        brightness = defbrightness(c,n,light,r,
          ambient)
```

Figure 18: The scene now including shadows.

5.6 Moving the screen

By defining a point at which the eye looks, you can move around the screen and always place it between the eye and this centre point. To do this a special coordinate system (u, v) is needed, as you want to place a two-dimensional screen in three-dimensional space. (See page 13).

```
#eyepoint
a = Vector(0,10000,0)
centre = Vector(0,-1000,0)
ambient = 0.05
width  = 1400
height = 1400

n = centre-(a)
p = a+(n*(0.5))
if n.x == 0:
    uS = Vector(1,0,0)
else:
    uy = 1
    ux = -uy*n.y/n.x
    uS = Vector(ux,uy,0)
vS = uS.cross(n)
uS=uS.normalize()
vS=vS.normalize()
pixel = p-(uS*(0.5*width))
pixel = pixel-(vS*(0.5*height))
s0 = 0
t0 = height
```

Then the function that loops over the pixels is adjusted,

```
for j in range(0,height):
        lowpixel = pixel
        for i in range(0,width):
                pixel = pixel+(uS)
                v = pixel-(a)
##Same as before

        pixel = lowpixel
        pixel= pixel+(vS)
```

The code after calculating `v` stays the same.

5.7 Planes and triangles by area

The following code is for the classes of `triangles` and `planes`. It uses the concepts outlined on page 15, just the first part is needed for the plane and both parts for the triangle. Both are defined using three points, the corners of the triangle or three points which lie inside the plane. The functions to calculate the reflected light ray were simply taken from the sphere class.

```
class areaTriangle:
        def __init__(self,ta,tb,tc,colour,constant):
                "Creates a triangle with 3 points and a colour"
                self.ta = ta
                self.tb= tb
                self.tc = tc
                self.colour = colour
                self.Sa=(self.tb-self.tc).length()
                self.Sb=(self.ta-self.tc).length()
                self.Sc=(self.tb-self.ta).length()
                self.Area1= self.Area(self.Sa,self.Sb,self.Sc)
                self.Normal= (self.tb-self.ta).cross(self.tc-self.ta)
                self.d = self.Normal.scalarp(self.ta)
                self.constant = constant
#by putting these here, I only calculate them once per triangle and not
   for every single pixel.
        def Normalvector(self,anything):
                "Returns the normal vector of the triangle"
                return self.Normal
        def IntersectionP(self,a,v,t):
                "Returns the intersection point, a = eyepoint,
                  v= light vector, t= factor (calculate
                  with Intersection function)"
                nV = v*t
                return(a+nV)
        def Area(self,Sa,Sb,Sc):
                "Calculates the area of a triangle, uses Herons Formula,
                  input is length of sides"
                s = 0.5* (Sa+Sb+Sc)
                A= sqrt(s*(s-Sa)*(s-Sb)*(s-Sc))
                return A
```

```
def Intersection(self,a,v):
        "Calculates t for intersection of the triangle,
          a = eye point v= direction of light ray"
        no = self.Normal
        d = self.d
        m = no.scalarp(v)
        if m == 0:
                return  None
        else:
                f = a.scalarp(no)
                t = (d-f)/m
                if t < 0:
                        return None

                else:
                        Sa,Sb,Sc= self.Sa,self.Sb,self.Sc
                        A1=int(self.Area1)
                        P = self.IntersectionP(a,v,t)
                        AP = self.ta-P
                        AP = AP.length()
                        BP = self.tb-P
                        BP = BP.length()
                        CP = self.tc-P
                        CP = CP.length()
                        A2 = self.Area(Sb,AP,CP)
                        A3 = self.Area(Sc,BP,AP)
                        A4 = self.Area(Sa,BP,CP)
                        pA = int(A2 + A3 + A4)
                        x= A1 - pA
                        if  x>= 0:
                                return t
                        else:
                                return None
def reflectedray(self,v,n):
        "Returns the reflected vector, v = light vector,
           n = normal vector"
        a = v.scalarp(n)
        b = n.scalarp(n)
        c = a/b
        p = n*2*c
        w = v-p
        return w
def tolight(self,intersection,light):
        "Returns the vector leading from the point of
          intersection to the light source"
        return light-(intersection)
```

```
class Plane:
        def __init__(self,ta,tb,tc,colour,constant):
                "Creates a plane defined by 3 points and a colour"
                self.ta = ta
                self.tb= tb
                self.tc = tc
                self.colour = colour
                self.Normal= (self.tb-self.ta).cross(self.tc-self.ta)
                self.d = self.Normal.scalarp(self.ta)
                self.constant = constant
        def Normalvector(self,s):
                "Returns the normal vector of the plane"
                return self.Normal
        def IntersectionP(self,a,v,t):
                "Returns the intersection point, a= eyepoint,
                   v= light vector, t= factor (calculate
                   with Intersection function)"
                nV = v*t
                return(a+nV)
        def Intersection(self,a,v):
                "Calculates t for intersection of the plane,
                  a = eyepoint v= direction of light ray"
                no = self.Normal
                d = self.d
                m = no.scalarp(v)
                if m == 0:
                        return  None
                else:
                        f = a.scalarp(no)
                        t = (d-f)/m
                        if t < 0:
                                return None
                        else:
                                return t
        def reflectedray(self,v,n):
                "Returns the reflected vector, v = light vector,
                  n = normal vector"
                a = v.scalarp(n)
                b = n.scalarp(n)
                c = a/b
                p = n*(2*c)
                w = v-(p)
                return w
```

```
def tolight(self,intersection,light):
        "Returns the vector leading from the point
          of intersection to the light source"
        return light-(intersection)
```

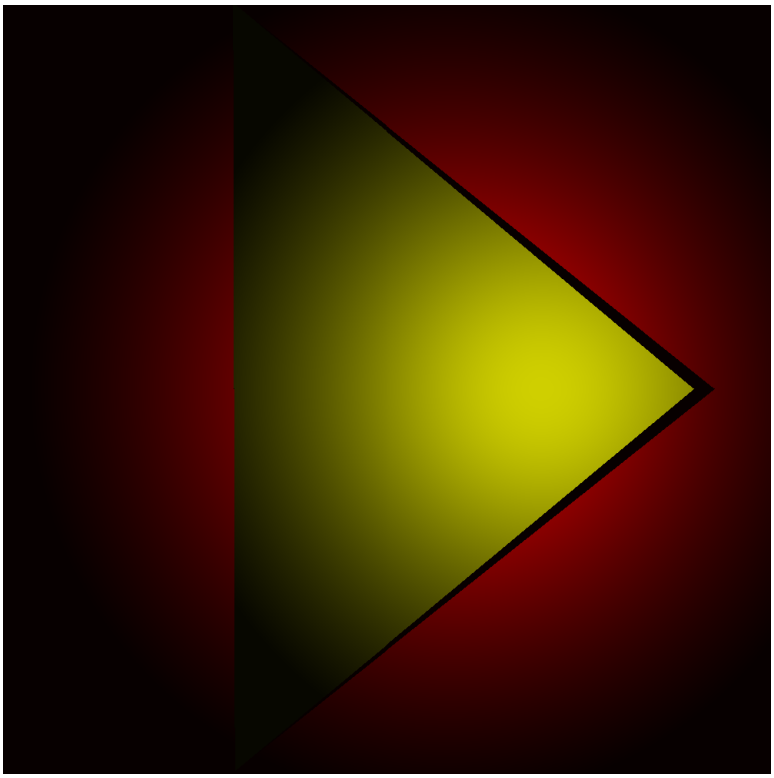


Figure 19: A triangle and a plane together in a scene.

5.8 Triangles again

The `insideout` function checks if a point that lies on the same plane as a polygon, lies inside or outside the polygon. It does this by taking two lists, the first list contains the vectors going from corner A to corner B, from corner B to corner C etcetera. The second list contains the vectors leading from corner A to the intersection point, from corner B to the intersection point etcetera. As described on page 17, it then makes sure that all the cross products point in the same direction, either the same or the opposite direction of the normal vector. The new class `Triangle` uses the inside outside test.

```
def insideout(cornercorner,cornerpoint,normal):
    "Used to check if a point lies inside or outside a polygon"
    " A list, a-b then b-c then c-d, going up to n-a,
       a list a-p, b-p, c-p, going up to n-p"
    "The two lists should be the same length"
    firstone = cornercorner[0].cross(cornerpoint[0])
    check = firstone.scalarp(normal)
    if check < 0:
        for i in range (1,len(cornercorner)):
            s = cornercorner[i]
            v = cornerpoint[i]
            n = s.cross(v)
            scal = n.scalarp(normal)
            if scal > 0:
                out = False
                break
            else:
                out = True
    else:
        for i in range (1,len(cornercorner)):
            s = cornercorner[i]
            v = cornerpoint[i]
            n = s.cross(v)
            scal = n.scalarp(normal)
            if scal < 0:
                out = False
                break
            else:
                out = True
    return out
```

```
class Triangle:
        def __init__(self,ta,tb,tc,colour):
                "Create a triangle with 3 points and a colour"
                self.ta = ta
                self.tb= tb
                self.tc = tc
                self.AB=(self.tb-(self.ta))
                self.BC=(self.tc-(self.tb))
                self.CA=(self.ta-(self.tc))
                self.Normal= (self.tb-self.ta).cross(self.tc-self.ta)
                self.d = self.Normal.scalarp(self.ta)
                self.colour= colour
#by putting these here, they are only calculated once per triangle
  and not for every single pixel.
        def Normalvector(self,anything):
                "Returns the normal vector of the triangle"
                return self.Normal
        def IntersectionP(self,a,v,t):
                "Returns the intersection point, a = eyepoint,
                   v= light vector, t= factor (calculate with
                  Intersection function)"
                nV = v*t
                return(a+nV)
        def Intersection(self,a,v):
                "Calculates t for intersection of the triangle,
                   a = eye point v= direction of light ray"
                no = self.Normal
                d = self.d
                m = no.scalarp(v)
                if m == 0:
                        return  None
                else:
                        f = a.scalarp(no)
                        t = (d-f)/m
                        if t < 0:
                                return None
                        else:
                                p = self.IntersectionP(a,v,t)
                                AB = self.AB
                                BC = self.BC
                                CA = self.CA
                                AP = p-self.ta
                                BP = p-self.tb
                                CP = p-self.tc
                                cornerc = (AB,BC,CA)
                                cornerp = (AP,BP,CP)
                                ins = insideout(cornerc,cornerp,no)
```

```
                        if ins == True:
                            return t
                        else:
                            return None
    def reflectedray(self,v,n):
            "Returns the reflected vector, v = light vector,
               n = normal vector"
            a = v.scalarp(n)
            b = n.scalarp(n)
            c = a/b
            p = n*2*c
            w = v-p
            return w
    def tolight(self,intersection,light):
            "Returns the vector leading from the point of
               intersection to the light source"
            return light-(intersection)
```

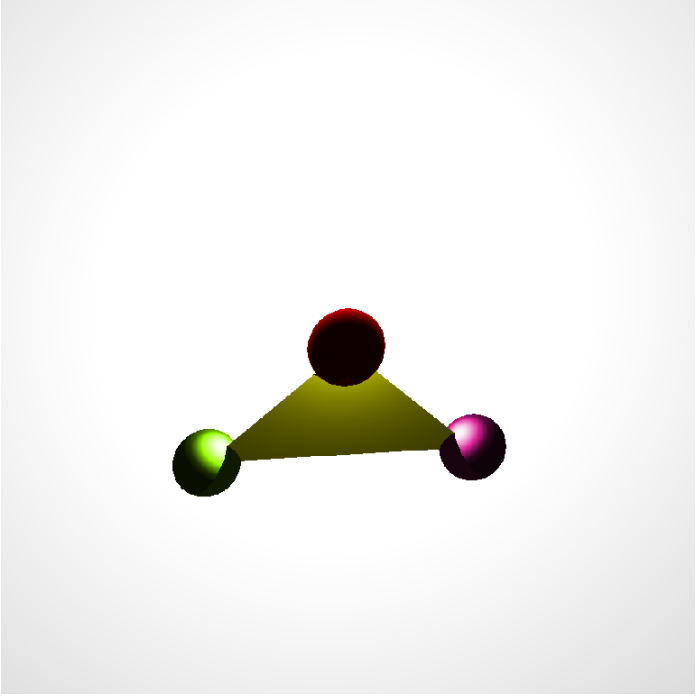


Figure 20: A scene with a triangle and three spheres centred at each of its corners.

5.9 Mirrors, multiple light sources

To have multiple light sources, you place them in a list and you iterate through them. For each you first check if there's an object between the point and the light source. If yes, you don't change the brightness as it would be covered by a shadow. If not, you calculate the highlights and the diffuse lighting (if these are bigger than 0, otherwise you just set it to 0). You then add these for all the light sources together and add them to the ambient. Then you have the final brightness.
To treat an object as a mirror, when creating it you either say `True` or `False`. If `False`, you proceed the same way as before. If `True` and a ray hits this object you send out a ray from the intersection point facing the direction of the reflected vector. To do this, a function `sendray` is used. This function sends a ray from a point (eye) in a certain direction (vector leading from the eye to the current pixel) and checks which object it impacts with. If this object is a mirror, it sends out a ray from the intersection point. Once it hits a non-mirror object or if the ray touches nothing, it determines the colour and brightness at this point and then draws the pixel in that colour.

```
def sendray(a,v,Objects,alllights,ambient,i,j,reflectc,origobj):
                "Sends out a ray through a pixel and checks what object
                  it impacts with, then determines  the colour and the
                  brightness for that pixel and draws that pixel"
                o,t= Objectsort(a,v,Objects)
                if o == None:
                    Colour = Color("black")
                    v = v.normalize()
                    luminance = 0
                    for light in alllights:
                        v2 = light-a
                        v2 = v2.normalize()
                        v = v.normalize()
                        brightness = backbrightness(v,v2)
                        luminance += brightness
                        if luminance >1:
                            luminance = 1
                            break
                    Colour.luminance = brightness
                    Pixel([i,j],Colour)
                else:
                    y = o.IntersectionP(a,v,t)
                    n = o.Normalvector(y)
                    n = n.normalize()
                    r = o.reflectedray(v,n)
                    r = r.normalize()
```

```
                    if o.mirror == True:
                        if reflectc < 100:
                            reflectc += 1
#amount of reflections allowed
                            objs = list(Objects)
                            objs.remove(o)
                            origobj = o
                            sendray(y,r,objs,alllights,ambient,i,j,
                              reflectc,origobj)
                        else:
                            brightness = defbrightness(o,y,n,alllights,r
                              ,Objects)
                            Colour = Color(o.colour)
                            Colour.luminance = brightness
                            Pixel([i,j],Colour)

                    else:
                        brightness = defbrightness(o,y,n,alllights,r
                          ,Objects)
                        Colour = Color(o.colour)
                        Colour.luminance = brightness
                        Pixel([i,j],Colour)
```

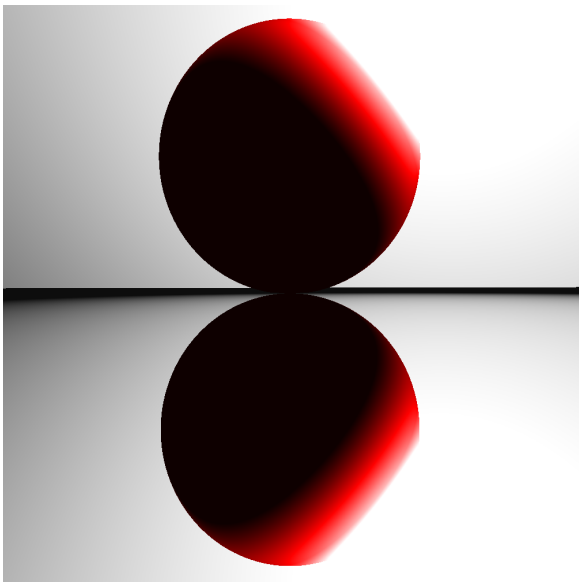


Figure 21: A sphere sitting on top of a mirror plane.

5.10 Roughness, materials

To give an object a certain roughness, you want to create a random variation in the brightness. Here a random number between 0 and 0.3 is used. Every object has a constant `roughness` which is multiplied with this random number and then subtracted from the brightness. We also can split up the constant c with which the brightness had been multiplied with, into a diffuse, ambient and highlight constant. Each component of the lighting will be multiplied with its own constant. Thus, you now can set the diffuse constant to 0 if you want an object with only the highlights and the ambient brightness.
To keep the code somewhat organized when creating objects, instead of having to add all these different variables to an object, you can define them in a list, containing:

- The diffuse constant
- The ambient constant
- The highlight constant
- The colour
- The roughness
- If it is a mirror or not

All classes have now been slightly changed, in a similar fashion to how the sphere was changed. `defbrightness` additionally has been updated to contain all of this.

```
import random
class Sphere:
        def __init__(self,m,r,mat):
                """Creates a Sphere with the centre m (a vector),
                  radius r and a material"""
                self.m=m
                self.r=r
                self.material = mat
                self.colour=mat[3]
                self.difcon=mat[0]
                self.ambcon=mat[1]
                self.higcon=mat[2]
                self.rough=mat[4]
                self.mirror = mat[5]
def defbrightness(o,intersectionp, n, l, r,ambient,Objects):
        "object, intersection point, the normal vector for the
          tangential plane, list of lights, reflected vector,
          ambient and objects."
        luminance = ambient * o.ambcon
        for lobj in l:
            light = o.tolight(intersectionp,lobj)
            sha= isshadow(light,intersectionp,Objects,o)
```

```
        if sha == False:
            light = light.normalize()
            h = highlights(r,light)  *o.higcon
            d = diffuse(n,light)  * o.difcon
            if d < 0:
                d = 0
            if h < 0:
                h = 0
            luminance = luminance + d + h
        if luminance > 1:
            luminance = 1
            break
luminance= lightfunction2(luminance)
roughness = random.uniform(0,0.3) * o.rough
luminance = luminance - roughness
return max(0,luminance)
```

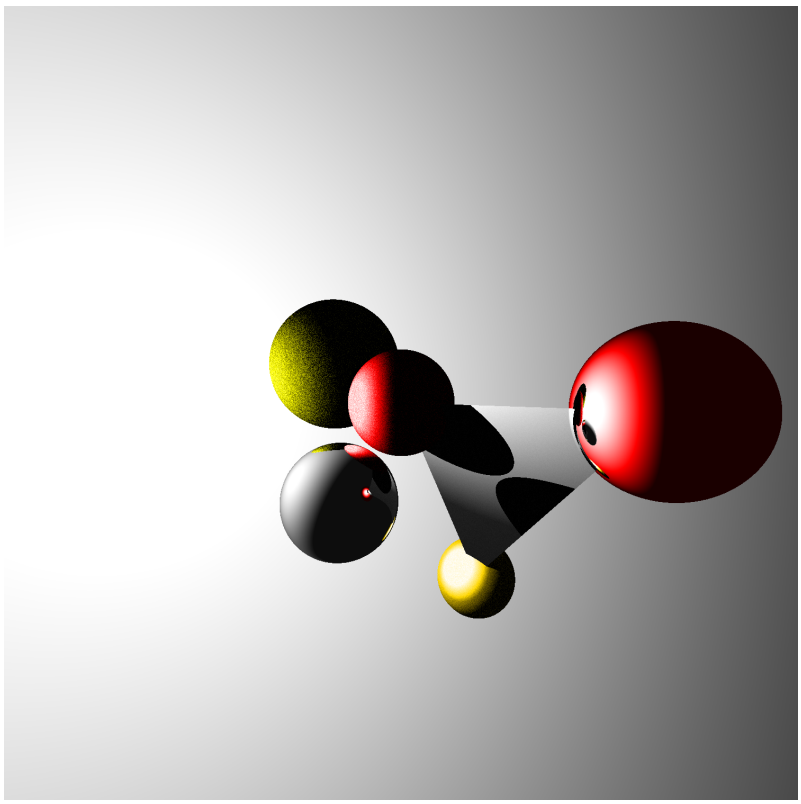


Figure 22: A scene containing two mirrors, one in the form of a triangle and one in the form of a sphere. Then three normal spheres and a light green sphere.

5.11 Moving objects

When creating a scene, it can be helpful if you have the option to shift/copy/reflect objects. The following functions are for triangles, but similar ones can be used for spheres and planes. `changenormal` is used for triangles and planes to change the direction the normal vector is facing, as the diffuse brightness depends on the normal vector and in some situations this might not be showing in the desired direction. `shift` shifts the object by a vector and `copy` creates a copy of the object that is shifted by a vector. Finally, `reflect` reflects the object over a point.

```
def changenormal(self):
        "Changes the direction the normal is showing"
        self.Normal= self.Normal*(-1)
        self.d = self.Normal.scalarp(self.ta)
def shift(self,factor):
        "shifts the triangle by a vector"
        self.ta = self.ta+factor
        self.tb = self.tb+factor
        self.tc = self.tc+factor
def copy(self,factor):
        "Copies and shifts the copy of the triangle by a vector"
        ta2= self.ta+factor
        tb2= self.tb+factor
        tc2= self.tc+factor
        return areaTriangle(ta2,tb2,tc2, self.material)
def reflect(self,point):
        "Reflects a triangle over a point"
        da = point - self.ta
        self.ta = self.ta + da *2
        db = point - self.tb
        self.tb = self.tb + db *2
        dc = point - self.tc
        self.tc = self.tc + dc *2
```

5.12 Rectangles and cuboids

The class `Rectangle` is created, this class is similar to the `Triangle` class and uses the `insideout` function to check if a point on the plane lies inside or outside the rectangle. The class `Cuboid` works by creating the 6 rectangles that make up the cuboid and adding them to the `objects` list.

```
class Rectangle:
        def __init__(self,ta,tb,tc,td,mat):
                "Create a rectangle with 4 points and a material"
                self.ta = ta
                self.tb= tb
                self.tc = tc
                self.td = td
                self.AB=self.tb-self.ta
                self.BC=self.tc-self.tb
                self.CD=self.td-self.tc
                self.DA=self.ta-self.td
                self.Normal= (self.tb-self.ta).cross(self.td-self.ta)
                self.d = self.Normal.scalarp(self.ta)
                self.material = mat
                self.colour=mat[3]
                self.difcon=mat[0]
                self.ambcon=mat[1]
                self.higcon=mat[2]
                self.rough=mat[4]
                self.mirror = mat[5]
#by putting these here, they are only calculated once per rectangle
  and not for every single pixel.
        def Normalvector(self,anything):
            "Returns the normal vector of the rectangle"
            return self.Normal
        def IntersectionP(self,a,v,t):
                "Returns the intersection point, a= eyepoint,
                  v= light vector, t= factor (calculate with
                  Intersection function)"
                nV = v*t
                return(a+nV)
        def Intersection(self,a,v):
                "Calculates t for intersection of the rectangle,
                   a = eye point v= direction of light ray"
                no = self.Normal
                d = self.d
                m = no.scalarp(v)
                if m == 0:
                        return  None
                else:
```

```
                f = a.scalarp(no)
                t = (d-f)/m
                if t < 0:
                        return None
                else:
                        p = self.IntersectionP(a,v,t)
                        AB = self.AB
                        BC = self.BC
                        CD = self.CD
                        DA =self.DA
                        AP = p-self.ta
                        BP = p-self.tb
                        CP = p-self.tc
                        DP = p-self.td
                        cornerc = (AB,BC,CD,DA)
                        cornerp = (AP,BP,CP,DP)
                        ins = insideout(cornerc,cornerp,no)
                        if ins == True:
                            return t
                        else:
                            return None
    def reflectedray(self,v,n):
            "Returns the reflected vector, v = light vector,
              n = normal vector"
            a = v.scalarp(n)
            b = n.scalarp(n)
            c = a/b
            p = n*(2*c)
            w = v-p
            return w
    def tolight(self,intersection,light):
            "Returns the vector leading from the point of
               intersection to the light source"
            return light-(intersection)
    def changenormal(self):
            "Changes the direction the normal is showing"
            self.Normal= self.Normal*(-1)
            self.d = self.Normal.scalarp(self.ta)
    def shift(self,factor):
            "shifts the whole rectangle by a vector"
            self.ta = self.ta+factor
            self.tb = self.tb+factor
            self.tc = self.tc+factor
            self.td = self.td+factor
    def copy(self,factor):
            "Copies the rectangle and shifts the copy by a vector"
            ta2= self.ta+factor
```

```
                tb2= self.tb+factor
                tc2= self.tc+factor
                td2= self.td+factor
                return Rectangle(ta2,tb2,tc2,td2, self.material)
        def reflect(self,point):
                "Reflects the rectangle over a point"
                da = point - self.ta
                self.ta = self.ta + da *2
                db = point - self.tb
                self.tb = self.tb + db* 2
                dc = point - self.tc
                self.tc = self.tc + dc* 2
                dd = point - self.td
                self.td = self.td + dd * 2
class Cuboid:
        def __init__(self,a,b,c,d,e,f,g,h,mat):
                "Creates 6 rectangles that are added to the object list"
                self.Sa = Rectangle(a,b,c,d,mat)
                self.Sb = Rectangle(a,b,e,f,mat)
                self.Sc = Rectangle(c,d,g,h,mat)
                self.Sd = Rectangle(g,h,e,f,mat)
                self.Se = Rectangle(a,d,g,f,mat)
                self.Sf = Rectangle(b,c,e,h,mat)
                Objects.append(self.Sa)
                Objects.append(self.Sb)
                Objects.append(self.Sc)
                Objects.append(self.Sd)
                Objects.append(self.Se)
                Objects.append(self.Sf)
```

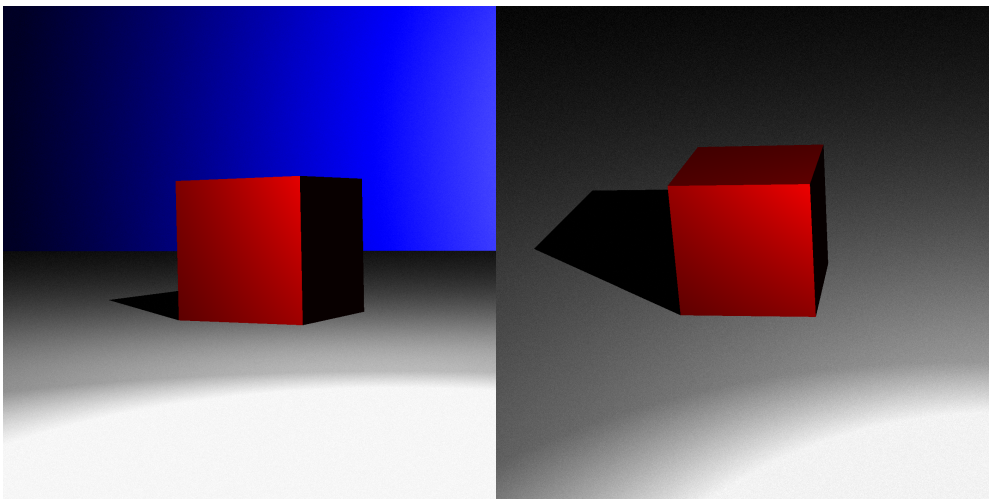


Figure 23: A cube lying on top of a plane, shown from two different angles.

5.13 Data format

By rendering the images with the library `canvas`, the challenge is that resulting images end up being very large (a few hundred megabytes). To avoid this, a different graphics tool can be used. Here `matplotlib` and `numpy` are used. First you need to import the packages. Then you create a function that turns a colour into its RGB equivalent. You then generate a two-dimensional matrix. So, every pixel is represented in this matrix with its height/width coordinates. The information stored in the matrix is the pixels colour. The function `pixel` is then changed so that you place the RGB code for the pixel [`i,j`] into the matrix. In the end you display the matrix by creating an image where every pixel has the colour depending on its RGB code.

```
import matplotlib.pyplot as plt
import numpy as np
def colbinfo(col):
        "Takes a colour and returns its RGB value"
        f=col.rgb
        g=[]
        for i in f:
          g.append(int(255*i+0.5))
        return g

def Pixel(p,col):
        i = p[0]
        j = p[1]
        frbrgb=farbinfo(col)
        image_matrix[i,j]=frbrgb
#Creates a two-dimensional matrix
image_matrix = np.ones( (width, height,3), dtype=np.uint8 )

#Storing the image
dotspi=300
grs=width/dotspi
fig=plt.figure(frameon=False, facecolor="red", edgecolor="blue")
fig.set_size_inches(grs,grs)
ax=plt.Axes(fig,[0,0,1.,1.])
ax.set_axis_off()
fig.add_axes(ax)
ax.imshow(image_matrix,aspect="auto")

##Saves the image
fig.savefig("Renderx.png", dpi=dotspi)
### Displays the image
plt.show()
```

6 Summary

In this paper I proceeded by first solving a problem mathematically and then writing the program. At the start, progress was slow as the basic blocks, like the class of vectors, had to be built up. Once I had these, I could proceed at a quicker rate and the code quickly went from being able to trace a sphere illuminated only by ambient lighting to being able to trace multiple spheres, all incorporating the different components of lighting while simultaneously casting shadows over each other. Then I added other objects like squares, triangles, and planes and the option to turn any object into a mirror . Following this, the option to control the components of lighting for every object separately was added, as well as the option to have multiple lights sources, each illuminating the scene with its own light, thus there were shadows of varying degrees depending on how many of the paths leading to the light sources were being blocked by different objects. Adding the choice of roughness of a given object allowed me to create various scenes with similar objects of the same colour. By playing around with the roughness and lighting constants of these, the objects would end up looking very different. One major problem I faced towards the end was the increasing time required to render the images (around 10-20 minutes for complex images) as the efficiency of the program was not one of the primary focuses for me. Nevertheless, the toolkit my program represents is extensive and can be used to render a large variety of images.

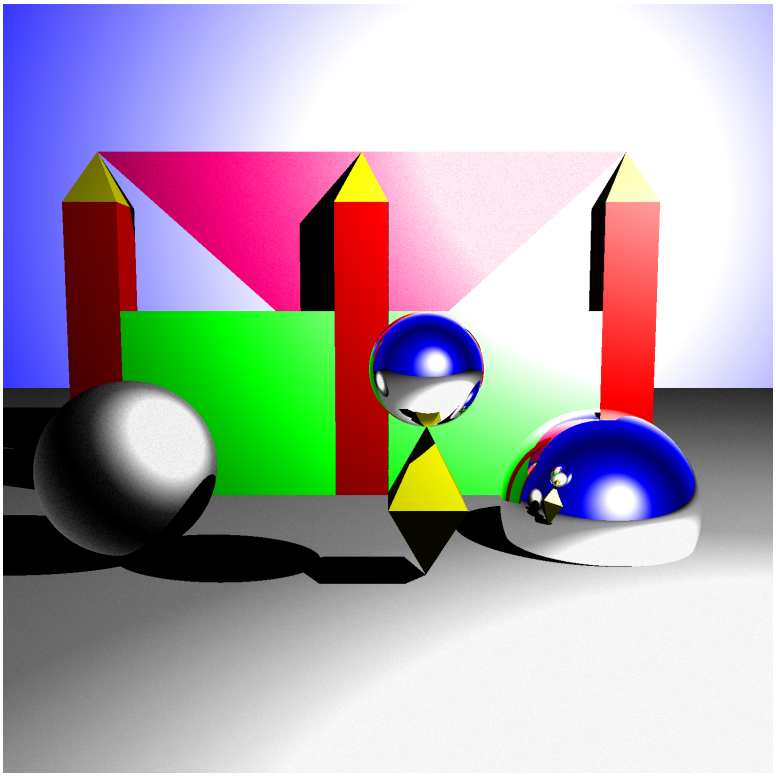


Figure 24: A scene depicting the usage of different tools that I created in this paper.

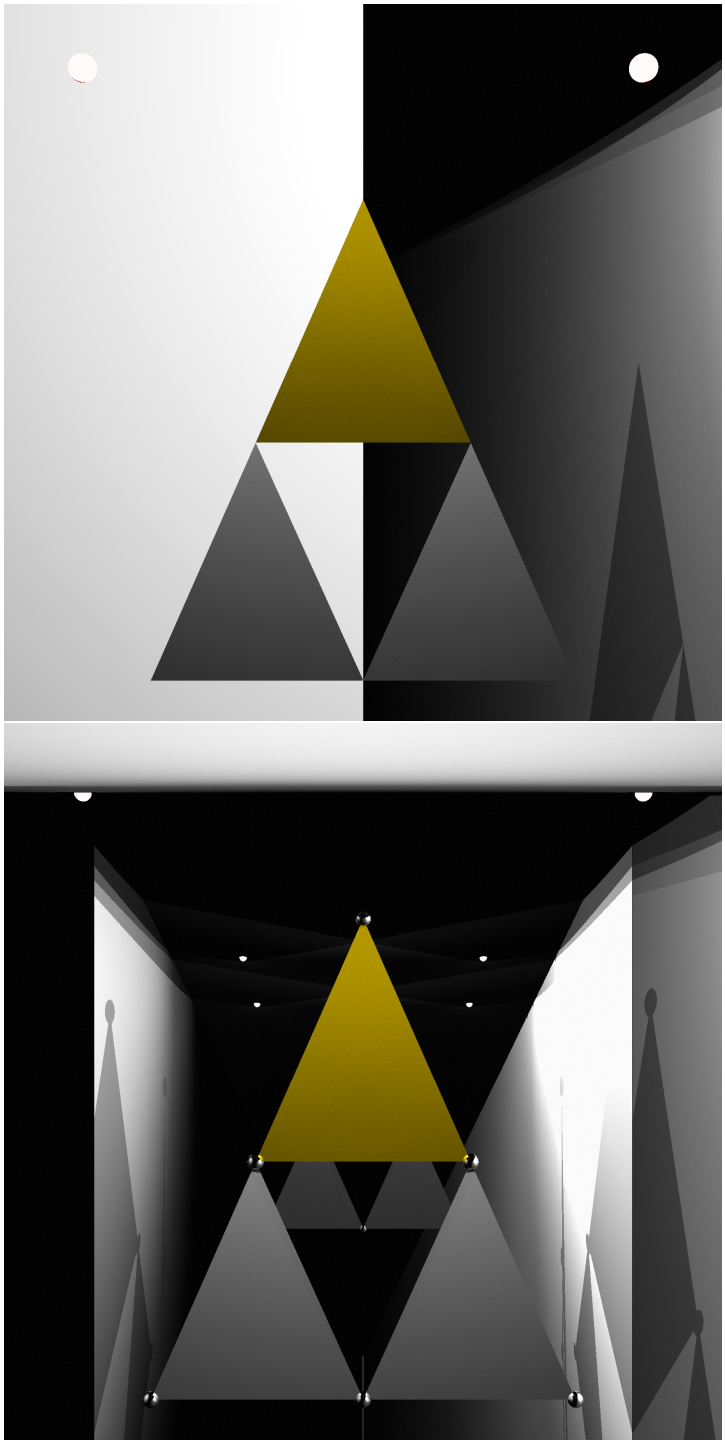


Figure 25: Another two images showcasing the usage of the different features.

7 Personal conclusion

I managed to get further than I had at first thought I would. At the beginning, I did not expect to be able to do things like the roughness of objects or turning them into mirrors. Working on this project has allowed me to see the use behind a lot of mathematics. I also learned a lot about the programming language python, as before I had never worked with the data structure classes and they were a vital part of this project, so I got to experiment a lot with them. Also, the usage of the python packages `numpy` and `matplotlib` to display the images was a new experience for me.

In the end, the code ended up getting rather long and keeping a clear overview of it was a tedious experience. But I was able to learn how to organize my code well and was made more aware of the importance of clearly documenting what each function does.
The work I have done for this paper has caught my interest and I would like to continue working on this code and adding additional features. For example, allowing a material to be permeable but also objects that can refract light like glass and water do, are challenges I did not have the time for yet would like to return to at a later stage. I would also like to further improve the efficiency of this program, as I am sure there are many things I could do a lot quicker than I did them here.

8 References

Internet references

https://en.wikipedia.org/wiki/ray_tracing 04.03.2017

https://en.wikipedia.org/wiki/Phong_shading 07.05.2017

https://www.mathsisfun.com/geometry/herons-formula.html 06.06.2017

http://www.wikilectures.eu/index.php/Lambert's_law 11.07.2017

Image references

Image 4: https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg 21.07.2017

9 Attachments

9.1 Final code

9.1.1 Classes and functions

These are all the classes and functions I created, that are later called to create and display objects.

```
from math import *
import math
import matplotlib.pyplot as plt
import numpy as np
from colour import *
import random
from matplotlib import *

def quadr_equation (a,b,c):
        "Returns the real answers for the equation ax^2+bx+c = 0"
        d = b**2-4*a*c
        if  d<0:
            #For this program, -1 can be used as an answer because for t
              we only take answers bigger than 0.
                return (-1,-1)
        elif  d == 0:
                return [-b/(2*a),-1]
        else:
                x1 = (-b+ sqrt(d))/(2*a)
                x2 = (-b- sqrt(d))/(2*a)
        return  [x1, x2]
class Vector:
        def __init__(self,x,y,z):
                self.x=x
                self.y=y
                self.z=z
        def __add__(self, other):
                """Additions operator + is overloaded."""
                return Vector(self.x+other.x,self.y+other.y,self.z+
   other.z)
        def __sub__(self, other):
                """Subtractions operator - is overloaded."""
                return Vector(self.x-other.x,self.y-other.y,self.z-other
   .z)
        def __mul__(self,t):
                """Multiplications operator * is overloaded."""
                return Vector(t*self.x,t*self.y,t*self.z)
```

```
        def components(self):
                """Returns a list of the components of the vector."""
                return [self.x, self.y, self.z]

        def scalarp(self, other):
                "Scalar product"
                return self.x*other.x+self.y*other.y+self.z*other.z
        def cross(self,other):
                "cross product"
                xcomp = self.y*other.z-self.z * other.y
                ycomp = self.z*other.x-self.x *other.z
                zcomp = self.x*other.y-self.y*other.x
                return Vector(xcomp,ycomp,zcomp)
        def length(self):
                "returns the length of the vector"
                return float((sqrt(self.scalarp(self))))
        def normalize(self):
                "Sets the vector to the length 1"
                r= self.length()
                return(self*(1/r))
class Sphere:
        def __init__(self,m,r,mat):
                """Creates a Sphere with the centre m (a vector), radius
    r and a material"""
                self.m=m
                self.r=r
                self.material = mat
                self.colour=mat[3]
                self.difcon=mat[0]
                self.ambcon=mat[1]
                self.higcon=mat[2]
                self.rough=mat[4]
                self.mirror = mat[5]
        def  Intersection(self,a,v):
                "Calculates t for intersection, a = eyepoint, v = light
   vector"
                m = self.m
                Qa = v.scalarp(v)
                b = 2*(a.scalarp(v)-m.scalarp(v))
                c = a.scalarp(a)+m.scalarp(self.m)-2*m.scalarp(a)-self.r
   **2
                t= quadr_equation(Qa,b,c)
                #The following returns the smaller, positive t value
                t1 = t[0]
                t2 = t[1]
                if t1 < 0:
                        if t2 < 0:
```

```
                                return None
                        else:
                                return t2
                else:
                        if t2 < 0:
                                return t1
                        elif t1 < t2:
                                return t1
                        else:
                                return t2
        def IntersectionP(self,a,v,t):
                "a = eyepoint, v= light vector, t= factor (calculate
with Intersection function)"
                nV = v*t
                return(a+nV)

        def Normalvector(self,s):
                "Returns the normal vector of the tangential plane to
the point of intersection (s)"
                return (s-self.m)

        def reflectedray(self,v,n):
                "Returns the reflected vector, v = light vector, n =
normal vector"
                a = v.scalarp(n)
                b = n.scalarp(n)
                c = a/b
                p = n*2*c
                w = v-p
                return w
        def tolight(self,intersection,light):
                "Returns the vector leading from the point of
intersection to the light source"
                return light-intersection
        def shifts(self,factor):
                "shifts the sphere by a vector"
                m2 = self.m + factor
                self.m = m2
        def copy(self,factor):
                "Creates a copy of the sphere that is shifted by a
vector"
                m2 = self.m + factor
                return Sphere(m2,self.r,self.material)
        def reflect(self,point):
            "Reflects the sphere over a point"
            d = point - self.m
            m2 = self.m + d* 2
```

```
            self.m = m2

class areaTriangle:
        def __init__(self,ta,tb,tc,mat):
                "Create a triangle with 3 points and a material"
                self.ta = ta
                self.tb= tb
                self.tc = tc
                self.Sa=(self.tb-self.tc).length()
                self.Sb=(self.ta-self.tc).length()
                self.Sc=(self.tb-self.ta).length()
                self.colour=mat[3]
                self.difcon=mat[0]
                self.ambcon=mat[1]
                self.higcon=mat[2]
                self.rough=mat[4]
                self.mirror = mat[5]
                self.Area1= self.Area(self.Sa,self.Sb,self.Sc)
                self.Normal= (self.tb-self.ta).cross(self.tc-self.ta)
                self.d = self.Normal.scalarp(self.ta)
#by putting these here, they are only calculated once per triangle and
   not for every single pixel.
        def Normalvector(self,anything):
                "Returns the normal vector of the triangle"
                return self.Normal
        def IntersectionP(self,a,v,t):
                "Returns the intersection point, a = eyepoint, v= light
   vector, t= factor (calculate with Intersection function)"
                nV = v*t
                return(a+nV)
        def Area(self,Sa,Sb,Sc):
                "Calculates the area of a triangle, uses Herons Formula,
    input is length of sides"
                s = 0.5* (Sa+Sb+Sc)
                A= sqrt(s*(s-Sa)*(s-Sb)*(s-Sc))
                return A
        def Intersection(self,a,v):
                "Calculates t for intersection of the triangle, a = eye
   point v= direction of light ray"
                no = self.Normal
                d = self.d
                m = no.scalarp(v)
                if m == 0:
                        return  None
                else:
                        f = a.scalarp(no)
                        t = (d-f)/m
```

```
                    if t < 0:
                            return None

                    else:
                            Sa,Sb,Sc= self.Sa,self.Sb,self.Sc
                            A1=int(self.Area1)
                            P = self.IntersectionP(a,v,t)
                            AP = self.ta-P
                            AP = AP.length()
                            BP = self.tb-P
                            BP = BP.length()
                            CP = self.tc-P
                            CP = CP.length()
                            A2 = self.Area(Sb,AP,CP)
                            A3 = self.Area(Sc,BP,AP)
                            A4 = self.Area(Sa,BP,CP)
                            pA = int(A2 + A3 + A4)
                            x= A1 - pA
                            if  x>= 0:
                                    return t
                            else:
                                    return None
    def reflectedray(self,v,n):
            "Returns the reflected vector, v = light vector, n =
normal vector"
            a = v.scalarp(n)
            b = n.scalarp(n)
            c = a/b
            p = n*2*c
            w = v-p
            return w
    def tolight(self,intersection,light):
        "Returns the vector leading from the point of intersection
to the light source"
        return light-(intersection)
    def changenormal(self):
            "Changes the direction the normal is showing"
            self.Normal= self.Normal*(-1)
            self.d = self.Normal.scalarp(self.ta)
    def shift(self,factor):
            "shifts the triangle by a vector"
            self.ta = self.ta+factor
            self.tb = self.tb+factor
            self.tc = self.tc+factor
    def copy(self,factor):
            "Copies and shifts the copy of the triangle by a vector"
            ta2= self.ta+factor
```

```
                tb2= self.tb+factor
                tc2= self.tc+factor
                return areaTriangle(ta2,tb2,tc2, self.material)
        def reflect(self,point):
                "Reflects a triangle over a point"
                da = point - self.ta
                self.ta = self.ta + da *2
                db = point - self.tb
                self.tb = self.tb + db *2
                dc = point - self.tc
                self.tc = self.tc + dc *2




class Triangle:
        def __init__(self,ta,tb,tc,mat):
                "Create a triangle with 3 points and a material"
                self.ta = ta
                self.tb= tb
                self.tc = tc
                self.AB=(self.tb-(self.ta))
                self.BC=(self.tc-(self.tb))
                self.CA=(self.ta-(self.tc))
                self.Normal= (self.tb-self.ta).cross(self.tc-self.ta)
                self.d = self.Normal.scalarp(self.ta)
                self.material = mat
                self.colour=mat[3]
                self.difcon=mat[0]
                self.ambcon=mat[1]
                self.higcon=mat[2]
                self.rough=mat[4]
                self.mirror = mat[5]
#by putting these here, they are only calculated once per triangle and
   not for every single pixel.
        def Normalvector(self,anything):
            "Returns the normal vector of the triangle"
            return self.Normal
        def IntersectionP(self,a,v,t):
                "Returns the intersection point, a= eye point, v= light
   vector, t= factor (calculate with Intersection function)"
                nV = v*t
                return(a+nV)
        def Intersection(self,a,v):
                "Calculates t for intersection of the triangle, a = eye
   point v= direction of light ray"
                no = self.Normal
```

```
                d = self.d
                m = no.scalarp(v)
                if m == 0:
                        return  None
                else:
                        f = a.scalarp(no)
                        t = (d-f)/m
                        if t < 0:
                                return None
                        else:
                                p = self.IntersectionP(a,v,t)
                                AB = self.AB
                                BC = self.BC
                                CA = self.CA
                                AP = p-self.ta
                                BP = p-self.tb
                                CP = p-self.tc
                                cornercorner = (AB,BC,CA)
                                cornerpoint = (AP,BP,CP)
                                ins = insideout(cornercorner,cornerpoint
,no)
                                if ins == True:
                                    return t
                                else:
                                    return None
        def reflectedray(self,v,n):
                "Returns the reflected vector, v = light vector, n =
normal vector"
                a = v.scalarp(n)
                b = n.scalarp(n)
                c = a/b
                p = n*2*c
                w = v-p
                return w
        def tolight(self,intersection,light):
            "Returns the vector leading from the point of intersection
to the light source"
            return light-(intersection)
        def changenormal(self):
                "Changes the direction the normal is showing"
                self.Normal= self.Normal*(-1)
                self.d = self.Normal.scalarp(self.ta)
        def shift(self,factor):
                "shifts the triangle by a vector"
                self.ta = self.ta+factor
                self.tb = self.tb+factor
                self.tc = self.tc+factor
```

```
        def copy(self,factor):
                "Copies and shifts the copy of the triangle by a vector"
                ta2= self.ta+factor
                tb2= self.tb+factor
                tc2= self.tc+factor
                return Triangle(ta2,tb2,tc2, self.material)
        def reflect(self,point):
                "Reflects a triangle over a point"
                da = point - self.ta
                self.ta = self.ta + da *2
                db = point - self.tb
                self.tb = self.tb + db *2
                dc = point - self.tc
                self.tc = self.tc + dc *2


class Rectangle:
        def __init__(self,ta,tb,tc,td,mat):
                "Create a rectangle with 4 points and a material"
                self.ta = ta
                self.tb= tb
                self.tc = tc
                self.td = td
                self.AB=self.tb-self.ta
                self.BC=self.tc-self.tb
                self.CD=self.td-self.tc
                self.DA=self.ta-self.td
                self.Normal= (self.tb-self.ta).cross(self.td-self.ta)
                self.d = self.Normal.scalarp(self.ta)
                self.material = mat
                self.colour=mat[3]
                self.difcon=mat[0]
                self.ambcon=mat[1]
                self.higcon=mat[2]
                self.rough=mat[4]
                self.mirror = mat[5]
#by putting these here, they are only calculated once per rectangle and
   not for every single pixel.
        def Normalvector(self,anything):
            "Returns the normal vector of the rectangle"
            return self.Normal
        def IntersectionP(self,a,v,t):
                "Returns the intersection point, a= eyepoint, v= light
   vector, t= factor (calculate with Intersection function)"
                nV = v*t
                return(a+nV)
        def Intersection(self,a,v):
```

```
            "Calculates t for intersection of the rectangle, a = eye
 point v= direction of light ray"
            no = self.Normal
            d = self.d
            m = no.scalarp(v)
            if m == 0:
                    return  None
            else:
                    f = a.scalarp(no)
                    t = (d-f)/m
                    if t < 0:
                            return None
                    else:
                            p = self.IntersectionP(a,v,t)
                            AB = self.AB
                            BC = self.BC
                            CD = self.CD
                            DA =self.DA
                            AP = p-self.ta
                            BP = p-self.tb
                            CP = p-self.tc
                            DP = p-self.td
                            cornercorner = (AB,BC,CD,DA)
                            cornerpoint = (AP,BP,CP,DP)
                            ins = insideout(cornercorner,cornerpoint
,no)
                            if ins == True:
                                return t
                            else:
                                return None
    def reflectedray(self,v,n):
            "Returns the reflected vector, v = light vector, n =
normal vector"
            a = v.scalarp(n)
            b = n.scalarp(n)
            c = a/b
            p = n*(2*c)
            w = v-p
            return w
    def tolight(self,intersection,light):
            "Returns the vector leading from the point of
intersection to the light source"
            return light-(intersection)
    def changenormal(self):
            "Changes the direction the normal is showing"
            self.Normal= self.Normal*(-1)
            self.d = self.Normal.scalarp(self.ta)
```

```
        def shift(self,factor):
                "shifts the whole rectangle by a vector"
                self.ta = self.ta+factor
                self.tb = self.tb+factor
                self.tc = self.tc+factor
                self.td = self.td+factor
        def copy(self,factor):
                "Copies the rectangle and shifts the copy by a vector"
                ta2= self.ta+factor
                tb2= self.tb+factor
                tc2= self.tc+factor
                td2= self.td+factor
                return Rectangle(ta2,tb2,tc2,td2, self.material)
        def reflect(self,point):
                "Reflects the rectangle over a point"
                da = point - self.ta
                self.ta = self.ta + da *2
                db = point - self.tb
                self.tb = self.tb + db* 2
                dc = point - self.tc
                self.tc = self.tc + dc* 2
                dd = point - self.td
                self.td = self.td + dd * 2

class Plane:
        def __init__(self,ta,tb,tc,mat):
                "Create a plane defined by 3 points which lie in the
   plane and a material"
                self.ta = ta
                self.tb= tb
                self.tc = tc
                self.Normal= (self.tb-self.ta).cross(self.tc-self.ta)
                self.d = self.Normal.scalarp(self.ta)
                self.material=mat
                self.colour=mat[3]
                self.difcon=mat[0]
                self.ambcon=mat[1]
                self.higcon=mat[2]
                self.rough=mat[4]
                self.mirror = mat[5]
#by putting these here, they are only calculated once per plane and not
   for every single pixel.
        def Normalvector(self,s):
            "Returns the normal vector of the plane"
            return self.Normal
        def IntersectionP(self,a,v,t):
```

```
            "Returns the intersection point, a= eyepoint, v= light
vector, t= factor (calculate with Intersection function)"
            nV = v*t
            return(a+nV)
    def Intersection(self,a,v):
            "Calculates t for intersection of the plane, a = eye
point v= direction of light ray"
            no = self.Normal
            d = self.d
            m = no.scalarp(v)
            if m == 0:
                    return  None
            else:
                    f = a.scalarp(no)
                    t = (d-f)/m
                    if t < 0:
                            return None
                    else:
                            return t
    def reflectedray(self,v,n):
            "Returns the reflected vector, v = light vector, n =
normal vector"
            a = v.scalarp(n)
            b = n.scalarp(n)
            c = a/b
            p = n*(2*c)
            w = v-(p)
            return w
    def tolight(self,intersection,light):
            "Returns the vector leading from the point of
intersection to the light source"
            return light-(intersection)
    def changenormal(self):
            "Changes the direction the normal is showing"
            self.Normal= self.Normal*(-1)
            self.d = self.Normal.scalarp(self.ta)
    def shift(self,factor):
            "shifts the plane by a vector"
            self.ta = self.ta+factor
            self.tb = self.tb+factor
            self.tc = self.tc+factor
    def copy(self,factor):
            "Copies the plane and shifts the copy by a factor"
            ta2= self.ta+factor
            tb2= self.tb+factor
            tc2= self.tc+factor
            return Plane(ta2,tb2,tc2, self.material)
```

```
        def reflect(self,point):
                "Mirrors the plane over a certain point"
                da = point - self.ta
                self.ta = self.ta + da *2
                db = point - self.tb
                self.tb = self.tb + db* 2
                dc = point - self.tc
                self.tc = self.tc + dc * 2

class Cuboid:
    def __init__(self,a,b,c,d,e,f,g,h,mat):
        "Creates 6 rectangles that are added to the object list"
        self.Sa = Rectangle(a,b,c,d,mat)
        self.Sb = Rectangle(a,b,e,f,mat)
        self.Sc = Rectangle(c,d,g,h,mat)
        self.Sd = Rectangle(g,h,e,f,mat)
        self.Se = Rectangle(a,d,g,f,mat)
        self.Sf = Rectangle(b,c,e,h,mat)
        Objects.append(self.Sa)
        Objects.append(self.Sb)
        Objects.append(self.Sc)
        Objects.append(self.Sd)
        Objects.append(self.Se)
        Objects.append(self.Sf)
def insideout(cornercorner,cornerpoint,normal):
    "Used to check if a point lies inside or outside a polygon"
    " A list, a-b then b-c then c-d, going up to n-a, a list a-p, b-p, c
   -p, going up to n-p"
    "The two lists should be the same length"
    firstone = cornercorner[0].cross(cornerpoint[0])
    check = firstone.scalarp(normal)
    if check < 0:
        for i in range (1,len(cornercorner)):
            s = cornercorner[i]
            v = cornerpoint[i]
            n = s.cross(v)
            scal = n.scalarp(normal)
            if scal > 0:
                out = False
                break
            else:
                out = True
    else:
        for i in range (1,len(cornercorner)):
            s = cornercorner[i]
            v = cornerpoint[i]
            n = s.cross(v)
```

```
            scal = n.scalarp(normal)
            if scal < 0:
                out = False
                break
            else:
                out = True
    return out
def Pixel(p,col):
        "Adds the colour information of a pixel to the image matrix"
        i = p[0]
        j = p[1]
        frbrgb=colinfo(col)
        image_matrix[i,j]=frbrgb

def highlights(refvector, lightvec):
        "Calculates the highlight component of the lighting, takes two
   normalized vectors, the reflected vector and the vector leading from
   the intersection point to the light source"
        r = refvector
        l = lightvec
        z = r.scalarp(l)
        if z < 0:
                return 0
        else:
                return z
def diffuse(normalvector,lightvec):
        "Calculates the diffuse component of the lighting, takes two
   normalized vectors, the normal vector for the plane and the vector
   leading from the intersection point to the light source"
        n = normalvector
        l = lightvec
        z = n.scalarp(l)
        if z < 0:
                return 0
        else:
                return z


def backbrightness(l,r):
        "Changes the brightness of the background pixel, needs two
   normalized vectors, the vector leading from the eye to the light and
   the vector going through the pixel"
        h = r.scalarp(l)
        if h < 0:
                return ambient
        elif h + ambient > 1 :
                return 1
```

```
        else:
                return h + ambient

def defbrightness(o,intersectionp, n, l, r,Objects):
        "Combines ambient, diffuse lighting and the highlights. Takes
    object (o), intersection point, n = the normalized normal vector for
    the plane, l = the list of light sources, r = the normalized
    reflected vector, Objects = the list of all objects."
        #ambcon/higcon/difcon are constants defined in the material of
    the object that allow the adjusting of the different light components
        luminance = ambient * o.ambcon
        for lobj in l:
            light = o.tolight(intersectionp,lobj)
            sha= isshadow(light,intersectionp,Objects,o)
            if sha == False:
                light = light.normalize()
                h = highlights(r,light)  *o.higcon
                d = diffuse(n,light)  * o.difcon
                if d < 0:
                    d = 0
                if h < 0:
                    h = 0
                luminance = luminance + d + h
                #once the brightness is maximum, there is no point in
    calculating anything additional as the number won't change.
            if luminance > 1:
                luminance = 1
                break
        luminance= lightfunction2(luminance)
        #Decides how rough every point of the object is.
        roughness = random.uniform(0,0.3) * o.rough
        luminance = luminance - roughness
        return max(0,luminance)

def isshadow(tolight, intersectionp, Objects,obj):
        "Checks if a point of an object lies in a shadow or not, warning
     tolight cannot be normalized!"
        z = list(Objects)
        z.remove(obj)
        o2,t = Objectsort(intersectionp,tolight,z)
        if t == None:
                return False
        elif t >= 1:
                return False
        else:
                return True
def lightfunction1(light):
```

```
        "light at 0.5 = 0.8"
        x = light
        b = -1.2* x * x + 2.2* x
        return b
def lightfunction2(light):
        "light at 0.7 = 0.5"
        x = light
        b = 0.476 * x * x+ 0.5239 * x
        return b
def Objectsort(a,v,objects):
        "Takes the light ray and checks which object, if any, it touches
    first. a = eye vector, v = direction of ray, objects = list of all
   the objects."
        tlist = []
        for i in objects:
                t = i.Intersection(a,v)
                tlist.append(t)
        loc = None
        vt = math.inf
        leng = len(tlist)
        for i in range(leng):
                value = tlist[i]
                if value != None:
                        if value < vt:
                                vt = value
                                loc= i
        if loc == None:
                return None,None
        else:
                return objects[loc],vt
def sendray(a,v,Objects,alllights,ambient,i,j,reflectc,origobj):
                "Sends out a ray through a pixel and checks what object
   it impacts with, then determines the colour and the brightness for
   that pixel and writes that information into the matrix"
                o,t= Objectsort(a,v,Objects)
                if o == None:
                    Colour = Color("black")
                    v = v.normalize()
                    luminance = 0
                    for light in alllights:
                        v2 = light-a
                        v2 = v2.normalize()
                        v = v.normalize()
                        brightness = backbrightness(v,v2)
                        luminance += brightness
                        if luminance >1:
                            luminance = 1
```

```
                            break
                    Colour.luminance = brightness
                    Pixel([i,j],Colour)
                else:
                    y = o.IntersectionP(a,v,t)
                    n = o.Normalvector(y)
                    n = n.normalize()
                    r = o.reflectedray(v,n)
                    r = r.normalize()
                    if o.mirror == True:
                        if reflectc < 100:
                            reflectc += 1
#amount of reflections allowed
                            objs = list(Objects)
                            objs.remove(o)
                            origobj = o
                            sendray(y,r,objs,alllights,ambient,i,j,
   reflectc,origobj)
                        else:
                            brightness = defbrightness(o,y,n,alllights,r
   ,Objects)
                            Colour = Color(o.colour)
                            Colour.luminance = brightness
                            Pixel([i,j],Colour)

                    else:
                        brightness = defbrightness(o,y,n,alllights,r,
   Objects)
                        Colour = Color(o.colour)
                        Colour.luminance = brightness
                        Pixel([i,j],Colour)
def colinfo(col):
    "Takes a colour and returns its RGB value"
    f=col.rgb
    g=[]
    for i in f:
        g.append(int(255*i+0.5))
    return g
```

9.1.2 Main body

The following is the main body. You first define what you want to display and how, it then generates the `u/v` coordinate system and then loops over all the pixels

```
#centre
centre = Vector(0,0,0)
#eyepoint = Vector(400,400,400)
object1 = #can be chosen
Objects = [object1]  #works with multiple objects as well
allllights = [] #can be filled with as many vectors as wanted, each one
    being a light source

#values
ambient = 0.05
#for the screen
width = 14000
height = 1400

#for the u/v coordinate system
n = centre-(a)
p = a+(n*(0.5))
if n.x == 0:
    uS = Vector(1,0,0)
else:
    uy = 1
    ux = -uy*n.y/n.x
    uS = Vector(ux,uy,0)
vS = uS.cross(n)
uS=uS.normalize()
vS=vS.normalize()
pixel = p-(uS*(0.5*width))
pixel = pixel-(vS*(0.5*height))

#Creates a two-dimensional matrix
image_matrix = np.ones( (width, height,3), dtype=np.uint8 )
# for percentage
per = 0
leng = 100/height
add = 15 * leng
count = 0

#main part, loops over all the pixels and draws them using the above
    functions
for j in range(0,height):
        if count == 15:
                per = per + add
                print(int(per),"%")
                count = 0
        count += 1
        lowpixel = pixel
        for i in range(0,width):
```

```
                pixel = pixel+(uS)
                v = pixel-(a)
                reflectc= 0
                origobj = None
                sendray(a,v,Objects,alllights,ambient,i,j,reflectc,
    origobj)
          pixel = lowpixel
          pixel= pixel+(vS)
```

9.1.3 Storing the image

The following displays the image and saves it as `renderx.png`.

```
#Storing the image
dotspi=300
grs=width/dotspi
fig=plt.figure(frameon=False, facecolor="red", edgecolor="blue")
fig.set_size_inches(grs,grs)
ax=plt.Axes(fig,[0,0,1.,1.])
ax.set_axis_off()
fig.add_axes(ax)
ax.imshow(image_matrix,aspect="auto")

##Saves the image
fig.savefig("Renderx.png", dpi=dotspi)

### Displays the image
plt.show()
```

10 Word of honour

FREIES GYMNASIUM ZÜRICH
gegründet 1888

Plagiatsreglement
Ehrenwort zur Maturitätsarbeit

(in die Maturitätsarbeit zu integrieren)

Originalarbeit

Ich erkläre, dass es sich bei der eingereichten schriftlichen Arbeit um eine von mir selbst und ohne unerlaubte Beihilfe verfasste Originalarbeit handelt.

Verweise auf Quellen

Ich erkläre, dass fremde Quellen (Originaltexte, Sekundärliteratur, Bilder, Tabellen usw.), die in der oben genannten Arbeit verwendet wurden, mit Quellenangaben versehen sind.

Plagiatsreglement

Ich bin mir bewusst, dass das von mir unterzeichnete Plagiatsreglement immer noch und in besonderer Weise für die Maturitätsarbeit Gültigkeit hat.

Massnahmen bei Plagiaten

Die im Plagiats-Reglement festgehaltenen Massnahmen in Fällen von Plagiarismus habe ich zur Kenntnis genommen.

Name: Dodgson

Vorname: Lucas

Ort, Datum: Horgen, 19.10.2017

Unterschrift Schülerin/Schüler: .....

Unterschrift der Vertragspartner: .....

Freies Gymnasium Zürich, Arbenzstrasse 19, Postfach, 8034 Zürich, T 043 456 77 77, F 043 456 77 78, www.fgz.ch