COMS30115 - Coursework 1
Raytracing

Carl Henrik Ek & Magnus Burenius

February 5, 2019

In this lab you will implement a Raytracer, which draws images of 3D scenes by tracing the light rays reaching the simulated camera. The lab is divided into several steps. To get something on the screen you first need to learn how to,

- Represent 3D scenes using triangular surfaces.

- Trace the ray of each pixel in the camera image into the scene.

- Compute ray-triangle intersections, to find out which surface a ray hit.

You will then be able to write a program that draws the image seen to the left in figure. Next, you will extend this program by adding,

- Camera motion.

- Simulation of light sources.

- Reflection of light for diffuse surfaces.

Disclaimer

This unit contains a lot of programming but its not a programming unit. Most likely you are all much better programmers than I am, this means that you should take my code only as a suggestion. If you feel that you want to do fancy `C++` stuff please do, just remember that going down the Object-Oriented route often makes code really hard to reuse and terribly inefficent (Java anyone[a]). As the saying goes *"You wanted a banana but what you got was a gorilla holding the banana and the entire jungle"*. [b] is very true for building Object Oriented projects. This was said in defence of functional languages which are wrong for all sorts of other reasons but lets not go there. Before I start a rant, all I mean, be creative and do not consider my code anything but a suggestion.

---

[a]Personally I consider using Java a crime against computers
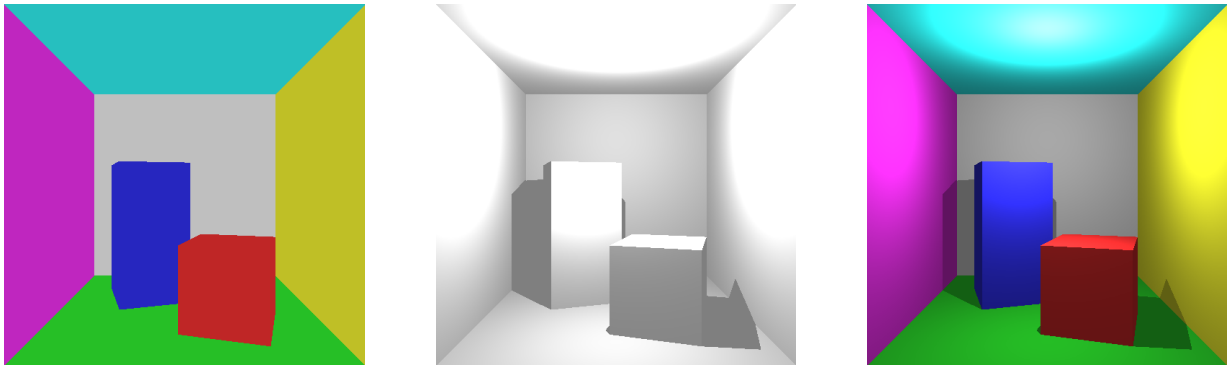[b]Joe Armstrong author of Erlang

Table 1: The output from this lab.

1 Representing Surfaces

In the first lab you coded a star-field by simulating a pinhole camera and its projection of moving 3D points. In this lab you will learn how to represent and draw surfaces, instead of points. The simplest surface to describe mathematically is a triangular surface. It is simple because it is planar and its geometrical properties can be described by only three 3D points: its vertices. For instance, from the three vertices we can compute the normal of the surface, i.e. a vector pointing out from the surface, being orthogonal to the surface itself. Although one can use other representations for surfaces triangles are the most used one for computer graphics due to its simplicity. See figure 1 for an example of a 3D model represented by triangular surfaces.
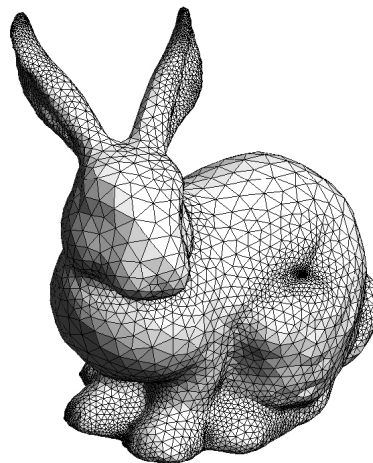

Figure 1: A 3D model represented by triangular surfaces. This bunny is a common test model in computer graphics. It can be found at: `http://graphics.stanford.edu/data/3Dscanrep`

When describing a surface for computer graphics it is not only the geometrical properties of the surface that are of interest. We are also interested in its appearance. The simplest way to describe the appearance of a triangle is with a single color. We will therefore use the following data structure to represent a triangle:

```
struct Triangle
{
    vec4 v0;
    vec4 v1;
    vec4 v2;
    vec4 normal;
    vec3 color;
};
```

Although the normal can be computed from the vertices, it can be convenient to have it in the data structure as well. Then the normals can be computed once and then stored so that we can easily access them later whenever we want, without any computations. We can now represent a 3D model by a set of triangles. To store all these triangles we will use the `stl::vector` template,

```
vector<Triangle> triangles;
```

To fill this vector with some triangles representing a 3D model you can use the function:

```
LoadTestModel( vector<Triangle>& triangles );
```

The function and the Triangle-struct is defined in the header file TestModel.h which you should include in the beginning of the code. This function will fill the vector it takes as an argument with the triangles describing the model seen in figure. This is a famous test model in computer graphics. You can read more about it at `http://www.graphics.cornell.edu/online/box/`. It is a cube shaped room with dimensions,

$$-1 \leq x \leq 1 \tag{1}$$
$$-1 \leq y \leq 1 \tag{2}$$
$$-1 \leq z \leq 1 \tag{3}$$

2 Intersection of Ray and Triangle

In our raytracer we want to trace rays, one for each pixel of the image, to see from where in the scene it originated from[1]. Since we use triangular surfaces to represent our model, this corresponds to finding the intersection between a triangle and a ray. We will now derive equations for this computation.

2.1 Triangles in 3D

Assume v_0, v_1, v_2 are real 4D homogenous vectors representing the vertices of the triangle, i.e. they are elements of $\mathbb{R}^4$. To describe a point in the triangle we construct a coordinate system that is aligned with the triangle. We let v_0 be the origin of the coordinate system and we let the axis be parallel to two of the edges of the triangle,

$$e_1 = v_1 - v_0 \tag{4}$$
$$e_2 = v_2 - v_0 \tag{5}$$

$e_1 \in \mathbb{R}^4$ is then a vector that is parallel to the edge of the triangle between v_0 and v_1. Similarly, $e_2 \in \mathbb{R}^4$ is parallel to the edge between v_0 and v_2. Any point $r \in \mathbb{R}^4$ in the plane of the triangle can then be described

---

[1]remember that we are following the light backwards

by two scalar coordinates $u \in \mathbb{R}$ and $v \in \mathbb{R}$, such that its position in 4D is,

$$r = v_0 + ue_1 + ve_2 \tag{6}$$

This holds for all points r that are in the plane of the triangle. For points that are not only in the same plane, but actually within the triangle we also have that,

$$0 < u \tag{7}$$
$$0 < v \tag{8}$$
$$u + v < 1 \tag{9}$$

2.2 Rays in 3D

Next, we need a way to describe points that are on a ray. By a ray we mean a segment of a line in 4D. We define a 4D line by a vector representing its start position $s \in \mathbb{R}^4$ and a vector representing its direction $d \in \mathbb{R}^4$. All points r lying on the line can then be written,

$$r = s + td \tag{10}$$

where $t \in \mathbb{R}$ is a scalar coordinate describing the position on the line. It is the signed distance, in units of $\|d\|$[fn::length of the vect, to the position r from the start position s. If we do not put any constrain on t we are describing a line that extends infinitely in both directions. However, by a ray we only mean the points that comes after the start position. For the ray we thus require,

$$0 \leq t \tag{11}$$

2.3 Intersection

To get the intersection between the plane of the triangle and the line of the ray we can insert the equation of the plane 6 into the equation of the line 10 and solve for the coordinates t, u, v,

$$v_0 + ue_1 + ve_2 = s + td \tag{12}$$

This is a equation of 4D vectors, but the only interesting part is in 3D as the homogenous coordinate have been removed. We thus have one equation for each vector component, i.e. three. And since we have three unknowns (t, u, v) it can be solved. To find the solution we re-formulate the equation using matrices, as one typically does when solving linear systems of equations,

$$v_0 + ue_1 + ve_2 = s + td \tag{13}$$
$$-td + ue_1 + ve_2 = s - v_0 \tag{14}$$
$$\begin{pmatrix} -d & e_1 & e_2 \end{pmatrix} \begin{pmatrix} t \\ u \\ v \end{pmatrix} = s - v_0 \tag{15}$$

This might still look a bit weird. To express the linear equation on the standard form we introduce the 3×3 matrix,

$$A = \begin{pmatrix} -d & e_1 & e_2 \end{pmatrix} \tag{16}$$

where the vectors $-d, e_1, e_2$ are the columns of the matrix. We also give a new name for our right hand side vector,

$$b = s - v_0 \tag{17}$$

And we put our unknowns in the vector,

$$x = \begin{pmatrix} t \\ u \\ v \end{pmatrix} \tag{18}$$

The linear equation 15 can then be written more concisely as,

$$Ax = b \tag{19}$$

And the solution x can be found by multiplying both sides with the inverse of the matrix A,

$$
\begin{aligned}
Ax &= b & (20)\\
A^{-1}Ax &= A^{-1}b & (21)\\
Ix &= A^{-1}b & (22)\\
x &= A^{-1}b & (23)
\end{aligned}
$$

Thus, if we can compute the inverse of the matrix A we can easily find the intersection point x. As we do not need the 4th coordinate at this time we will reduce the representation and do this directly in cartesian coordinates rather than homogenous. Luckily, `GLM` both has a class to represent 3×3 matrices (`glm::mat3`) and a function to compute matrix inverses. `GLM` has also used the possibility of C++ to overload operators. It has defined the operators `+,-,*,/` so that they can be used directly with `glm::vec3` and `glm::mat3`. We can therefore do the intersection computations by writing,

```
using glm::vec4;
using glm::mat4;
using glm::vec3;
using glm::mat3;

vec4 v0 = triangle.v0;
vec4 v1 = triangle.v1;
vec4 v2 = triangle.v2;

vec3 e1 = vec3(v1.x-v0.x,v1.y-v0.y,v1.z-v0.z)
vec3 e2 = vec3(v2.x-v0.x,v2.y-v0.y,v2.z-v0.z)
vec3 b = vec3(s.x-v0.x,s.y-v0.y,s.z-v0.z);

mat3 A( -d, e1, e2 );
vec3 x = glm::inverse( A ) * b;
```

Now we know the coordinates t, u, v for the intersection point. We can then check the inequalities 7, 8, 9, 11 to see whether the intersection occurred within the triangle and after the beginning of the ray. You Should write a function that does this,

```
bool ClosestIntersection(
    vec4 start,
    vec4 dir,
    const vector<Triangle>& triangles,
    Intersection& closestIntersection );
```

It takes the start position of the ray and its direction and a `std::vector` of triangles as input. It should then check for intersection against all these triangles. If an intersection occurred it should return true. Otherwise false. In the case of an intersection it should also return some information about the closest intersection. Define the following data structure to store this information,

```
struct Intersection
{
    vec4 position;
    float distance;
    int triangleIndex;
};
```

The argument closestIntersection sent to the function should be updated with the 4D position of the closest intersection. Its distance from the start of the ray and the index of the triangle that was intersected. When writing the function ClosestIntersection you might need to know the largest value a float can take. You can get this by including the header file "limits" in the beginning of your code and then writing,

```
float m = std::numeric_limits<float>::max();
```

If you are into linear algebra a good optional exercise is to calculate a closed form solution for the inverse of the matrix, instead of just using `glm::inverse`. This can be done with Cramer's rule. We need this result if we would like to increase the speed of the intersection computations, which will be the bottleneck of our program. Then, when we have the closed form solution we can postpone the computation of some of the matrix elements, until they are needed. We start by just computing the row needed to compute the distance t. Then, if t is negative there is no need to continue analyzing that intersection, as it will not occur. However, doing this is not necessary to pass the lab.

3 Tracing Rays

Now that you have a function that takes a ray and finds the closest intersecting geometry you have all you need to start rendering an image with raytracing. The idea is that you trace a ray for each pixel of the image. You then need to compute for each pixel what ray it corresponds to.

Initially we assume that the camera is aligned with the coordinate system of the model with x-axis pointing to the right and y-axis pointing down and z-axis pointing forward into the image. The start position for all rays corresponding to pixels is the camera position. A vector $d \in \mathbb{R}^4$ pointing in the direction where the light reaching pixel (x,y) comes from can then be computed as,

$$d = (x - W/2, y - W/2, f) \tag{24}$$

where W and H is the width and height of the image and f is the focal length of the camera, measured in units of pixels. We strongly encourage you to draw a figure of the pinhole camera and make sure that you understand this equation. The focal lenght can be thought of as the distance from the hole in the pinhole camera to the image plane. Add the following variables to store the camera parameters,

```
float focalLength = ...;
vec4 cameraPos( ..., ..., ... , 1.0);
```

Fill in ... with good values for these variables. Make sure that the model can be seen from the camera position you choose. What will the horizontal and vertical field of view be for the focal length you have chosen?

You are now ready to write the Draw function. In it you should loop through all pixels and compute the corresponding ray direction. Then call the function ClosestIntersection to get the closest intersection in that direction. If there was an intersection the color of the pixel should be set to the color of that triangle. Otherwise it should be black. The result should be similar to figure 2.
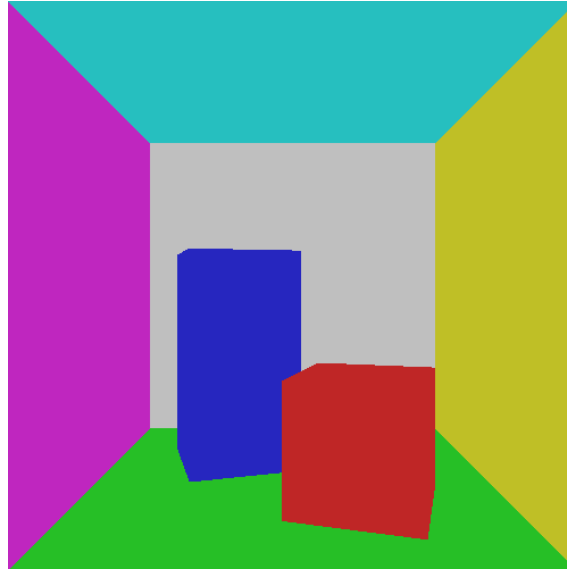
Figure 2: Tracing a light ray for each pixel to see which surface it first intersect.

4 Moving the Camera

We now have a simple visualization of our 3D scene. However, it would be nice if we could move the camera around. Doing so is good first because its just looks better and secondly because its the best way to debug stuff. Think about your rendering engine as function from a parameter space[2] to an image, by moving the camera around you alter pretty much every input parameter to your engine and this allows you to see where it breaks. First try to implement translation of the camera by updating the variable `cameraPos` if the user presses the arrow keys. You can use `SDL` to check if a key is pressed in this way,

---

[2]the space of vertices, camera paramters, colours, etc.

```
bool Update()
{
  // Compute frame time:
  static int t = SDL_GetTicks()
  int t2 = SDL_GetTicks();
  float dt = float(t2-t);
  t = t2;

  cout << "Render time: " << dt << " ms." << endl;

  SDL_Event e;
  while(SDL_PollEvent(&e))
    {
      if (e.type == SDL_QUIT)
        {
          return false;
        }
      else
        if (e.type == SDL_KEYDOWN)
          {
            int key_code = e.key.keysym.sym;
            switch(key_code)
              {
              case SDLK_UP:
                /* Move camera forward */
                break;
              case SDLK_DOWN:
                /* Move camera backwards */
                break;
              case SDLK_LEFT:
                /* Move camera left */
                break;
              case SDLK_RIGHT:
                /* Move camera right */
                break;
              case SDLK_ESCAPE:
                /* Move camera quit */
                return false;
              }
          }
    }
  return true;
}
```

Now you can use this information to translate the camera. Since raytracing is slow you probably want to make sure that you get this working first before you throw all your computation at it. One idea is to debug this in a really low resolution first. There are two versions of the Cornellbox included `TestModel.h` which is in cartesian coordinates and `TestModelH.h` which is in homogenous coordinates.

When you got that working you should add a variable that controls the rotation of the camera. Use a `glm::mat4` to represent the rotation matrix of the camera,

Code

```
mat4 R;
```

Then make the left and right arrow keys rotate the camera around the y-axis. It might be convinient to add another variable that stores the angle which the camera should be rotated around the y-axis,

Code

```
float yaw;
```

When the left and right arrow keys are pressed this angle should be updated and then the rotation matrix R should be updated to represent a rotation around the y-axis with this angle. If you do not remember how such a matrix looks you can have a look in your linear algebra book or at Wikipedia.

If the camera is rotated by the matrix R then vectors representing the right (x-axis), down (y-axis) and forward (z-axis) directions can be retrieved as,

Code

```
vec4 right(   R[0][0], R[0][1], R[0][2], 1 );
vec4 down(    R[1][0], R[1][1], R[1][2], 1 );
vec4 forward( R[2][0], R[2][1], R[2][2], 1 );
```

To model a rotating camera you need to use these directions both when you move the camera and when you cast rays. Implement this.

One thing that might be useful to implement is a LookAt function, so that you can position the camera somewhere in space but always make it "look" at the same position. There is a good explanation of this here LookAt function. As mentioned previously, you probably need to set a low width and height of the image for the algorithm to be fast enough for real-time motion. Since raytracing is slow it is typically not used for games and other interactive real-time applications. It is mainly used to render single images and special effects for movies. In lab 3 we will implement another algorithm, rasterization, which is faster and thus more suitable for interactive applications. After you got the camera motion working you can change back to a higher image resolution when doing the rest of the lab.

5 Illumination

You should now have a raytracer in which you can move the camera around. However, so far all points of a surface has exactly the same color. To add some more realism to the image we will add a light source. We will model an omni light. Such a light spreads light equally in all directions from a single point in space. The light can be defined by two variables: its position and power for each color component,

```
Code

vec4 lightPos( 0, -0.5, -0.7, 1.0 );
vec3 lightColor = 14.f * vec3( 1, 1, 1 );
```

The color vector describes the power P, i.e. energy per time unit E/t of the emitted light for each color component. Each component thus has the physical unit $W = J/s$. The light will spread uniformly around the point light source, like an expanding sphere. Therefore, if a surface is further away from the light source it will receive less light per surface area. To derive a formula for the received light at a distance r from the light source we can think about a sphere with radius r centered at the light source. The emitted light will be spread uniformly over the whole surface area of the sphere which is,

$$A = 4\pi r^2 \tag{25}$$

The power per area B reaching any point on the sphere is therefore,

$$B = \frac{P}{A} = \frac{P}{4\pi r^2} \tag{26}$$

When working with illumination this physical quantity is often used. Since it describes power per area it has the unit $W/m^2 = Js^{-1}m^{-2}$. Equation 26 describes the power per area reaching any surface point directly facing the light source. However, most of the time we will have surfaces that do not directly face the light source. Let $\hat{n}$ be a unit vector describing the normal pointing out from the surface and let $\hat{r}$ be a unit vector describing the direction from the surface point to the light source. Then if B be is the power of light reaching a virtual surface area directly facing the light source at the surface point, then the power per real surface D will just be a fraction of this,

$$D = B \max(\hat{r} \cdot \hat{n}, 0) = \frac{P \max(\hat{r} \cdot \hat{n}, 0)}{4\pi r^2} \tag{27}$$

We get the fraction between D and B by the projection of $\hat{r}$ on $\hat{n}$, i.e. a scalar product of two unit vectors. Since we do not allow negative light we take the max of the projected value and zero. If the angle between the surface normal and direction to the light source is larger than 90 degrees it does not receive any direct illumination.

One way to think about this projection factor is to consider light as a stream of particles. If the particles hit the surface from an angle, instead of straight ahead, they will spread out more over the surface and less particles will hit a fixed area over a fixed time. If it is raining and you place a bucket outside it will fill more quickly if it is raining straight from above than if the wind is blowing and the rain comes from the side.

Now we know how much light that reaches any surface point directly from the light source, i.e. without reflection via another surface. You should then implement the function:

```
Code

vec3 DirectLight( const Intersection& i );
```

It takes an intersection, which gives us the position where we want to know the direct illumination, as well as the index to the triangle that should get illuminated, therefore we also know the normal of the surface. The function should then return the resulting direct illumination described by equation 27.

Modify the Draw function such that it uses the function DirectLight to compute the color for every pixel, after you have computed the intersection it corresponds to. You should then get the result of figure 3. Then you should also make it possible to move the light source, using the keys W and S to translate it forward
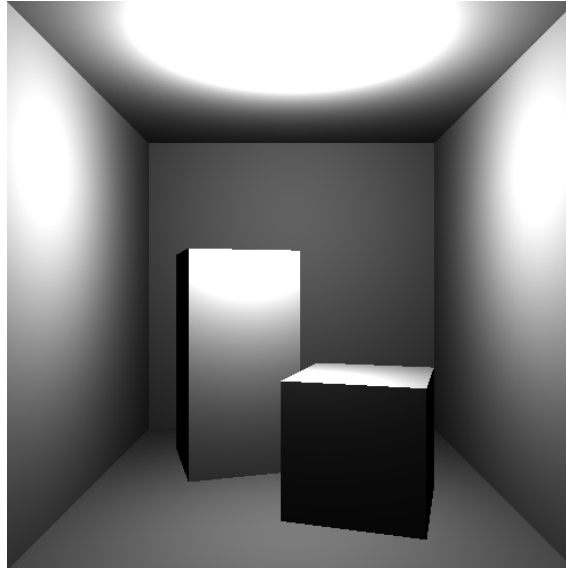
Figure 3: The incoming light.

and backward and A and D to translate it left and right and Q and E to translate it up and down. Do this in the Update function by writing something like,

```
Code

if( keystate[SDLK_w] )
    lightPos += forward;
```

However, this result does not correspond to how we would perceive the surface in the real world. It just tells how much light that reaches a surface point. What we actually see is the fraction of this that gets reflected in the direction of the ray. We thus need a model for this.
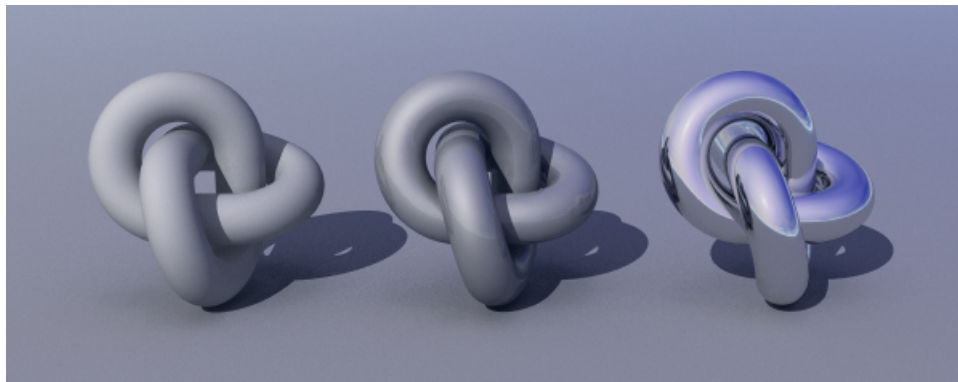

Figure 4: The left object is ideally diffuse. The right object is ideally specular. The middle object is somewhere in between.

We will model all surfaces as ideally diffuse (see figure 4). Such surfaces are simple to simulate because they reflect light equally in all directions. The viewing angle does not influence the percieved color of such a surface point. It will look the same from all directions, having no specular high lights. In that sense, it is the opposite of a mirror, which will look completely different depending on the viewing angle. A mirror has an ideally specular surface (see figure 4. Let $\rho = (\rho_r, \rho_g, \rho_b)$ describe the fraction of the incoming light that

gets reflected by the diffuse surface for each color component. Then the light that gets reflected is,

$$R = \rho * D = \frac{\rho * P \max(\hat{r} \cdot \hat{n}, 0)}{4\pi r^2} \tag{28}$$

where the operator $*$ denotes element-wise multiplication between two vectors. If we use `glm::vec3` for 3D vectors we can compute this element-wise multiplication by just writing,

```
Code

vec3 A(1,2,3);
vec3 B(4,5,6);
vec3 C = A*B; // C will be (4,10,18)
```

since `GLM` has overloaded the operator $*$ for two `glm::vec3` to represent element-wise multiplication. Extend the Draw-function such that only the reflected light gets drawn. The color stored in each triangle is its ρ. This should give the result seen in figure 5.

5.1 Direct Shadows

So far a surface gets no direct illumination if it does not face the light source. However, we would also like to simulate shadows if another surface intersects the ray from the light source to the surface. To simulate this we can use the ClosestIntersection function that we use to cast rays. When illuminating a surface point we cast another ray from it to the light source. Then we check the distance to the closest intersecting surface. If that is closer than the light source the surface is in shadow and does not receive any direct illumination from the light source. Add this to your DirectLight function. If it works you should see shadows like in figure 6.

5.2 Indirect Illumination

Just implementing the direct illumination will make some surface points appear pitch black. It only simulates one bounce of light from the light source via a surface point to the camera. In the real world light keeps reflecting multiple times and therefore reaches all surfaces at some point. At each reflection some of the light is absorbed by the surface, as specified by its ρ. This indirect illumination results in the smooth shadows that we see in the real world.

It is very computationally expensive to simulate the multiple bounces of light required for accurate computation of indirect illumination. This might be something that you can think of as an extension though. Instead, we will use a very simple approximation to model the indirect illumination. We will just assume that the indirect illumination is the same for every surface of the scene. Let N be the power per surface area of this constant indirect illumination. Then the total incident illumination of a surface point is,

$$T = D + N \tag{29}$$

And the reflected light is,

$$R = \rho * T = \rho * (D + N) \tag{30}$$

First create a new variable to store the constant approximation of the indirect illumination,

```
Code

vec3 indirectLight = 0.5f*vec3( 1, 1, 1 );
```

Then implement this illumination model, equation 30, in your Draw function. You should not alter the function DirectLight as it should only return the direct illumination which it already does. The result is an image were no surface is completely black, like in figure 7.
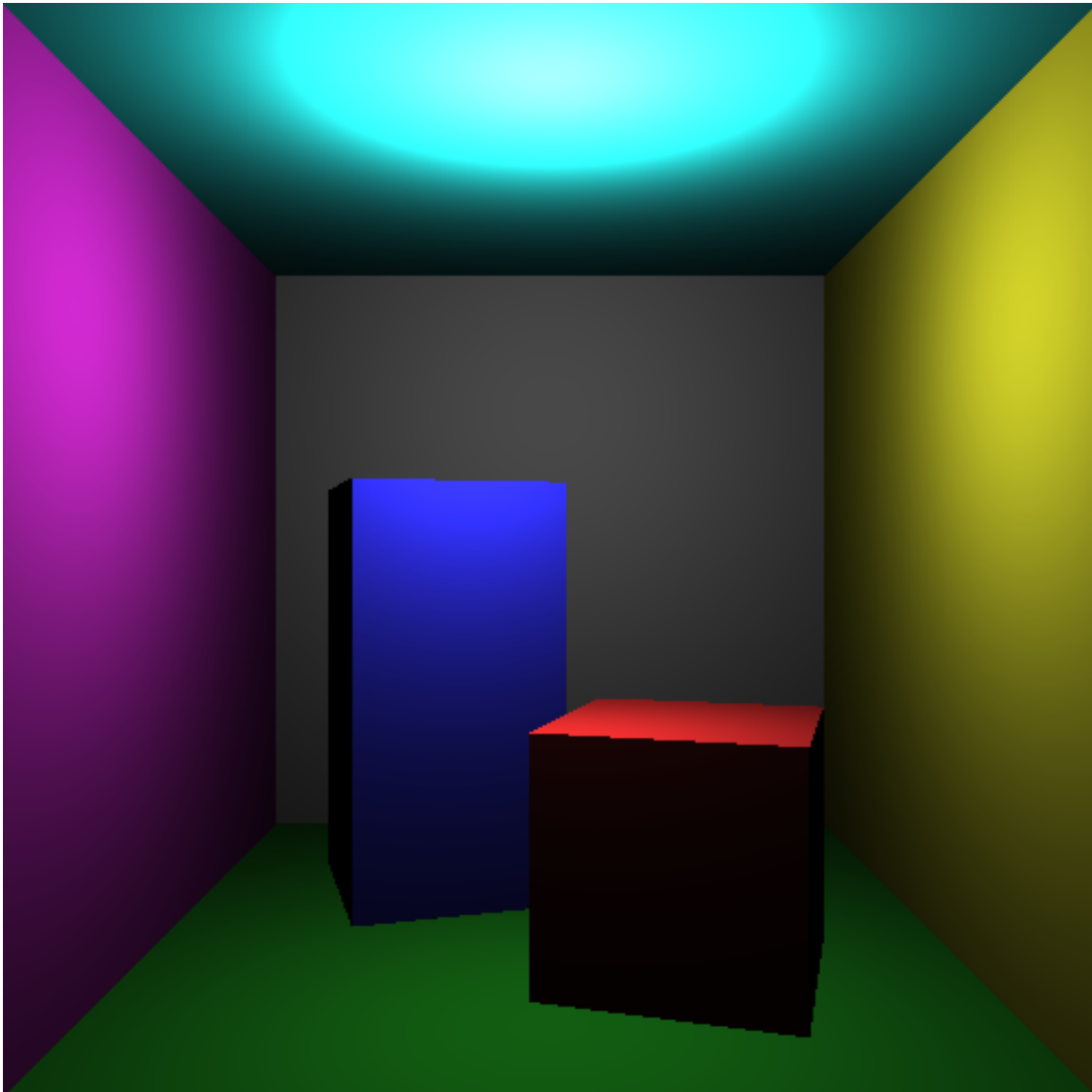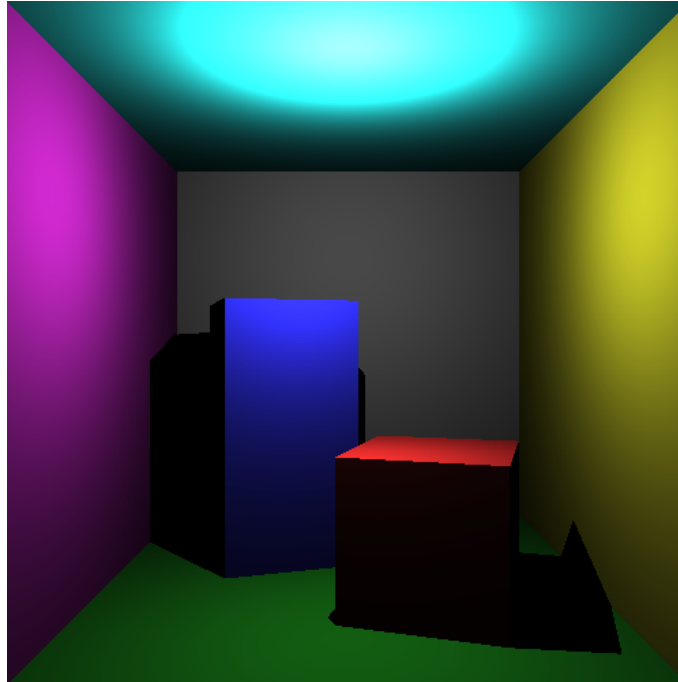
Figure 5: Direct illumination, without shadows.

Figure 6: Direct illumination with shadows.

6 Extensions

You have now written a raytracer, that handles direct illumination, shadows and approximates indirect illumination with a constant term. Now you have the oppurtunity to extend the raytracer in a manner of your choice. Try to think of something that you enjoy doing yourself and then extend it in that manner. If you want you could then add things like,

- Soft edges (anti-aliasing) by sending multiple slightly perturbed rays for each pixel.

- Depth of field, by modeling the camera lens. Includes soft edges.

- Indirect illumination by multiple bounces of light.

- Textures. Can be used for color/reflectance and/or normalmaps, and/or parallax mapping.

- Loading of general models.

- Storing the geometry in a hierarchical spatial structure for faster intersection queries. This speeds up the rendering which is good for more complex/interesting scenes.

- Specular materials. For example you could simulate water with normal maps.

- Using fractals to automatically generate geometry and textures for mountains, clouds and water.

- Optimisations: a raytracer is incredibly parallell and you can exploit to improve speed.
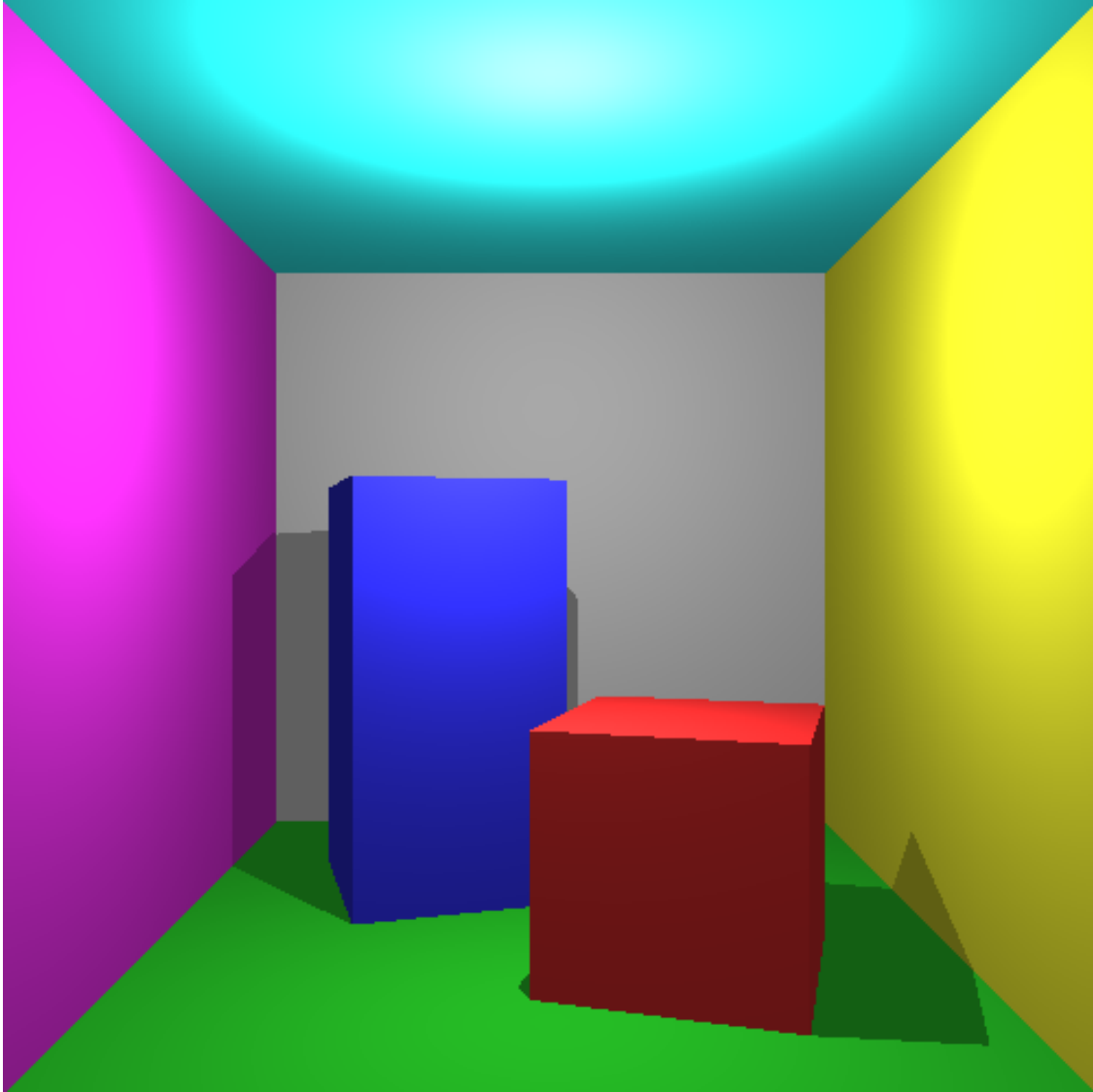
- Include elements from global illumination

- etc.

Figure 7: Direct illumination and constant approximation of indirect illumination.

The grade that you get for your extension depends on two things, first the challenge involved in understanding and implementing the extension but also the execution of the extension. I am really keen to see what your thoughts are and what you are planning to do so come and have a chat with me and I will be able to give you an indication of the credits that different things will give you. Dare to be inovative and follow your own interest I am sure we somehow can make computer graphics out of it.

Good Luck and Happy Hacking