Master Project
Computer Graphics: Rendering Track

Alhajras Algdairy
Advisor: Prof. Dr.-Ing. Matthias Teschner

September 1, 2021

Acknowledgment

Abstract

This report investigates four different acceleration data-structure methods for implementing a simple raytracer. These methods are KD-tree, Uniformgrid, BVH, and LBVH. The report aims to compare the different approaches and their impact on the raytracer rendering time. The raytracer used in this project builds upon a previous lab project where the foundation of the raytracer is already implemented; however, without using a sophisticated data-structure to enhance the rending performance; hence the focus of this report will be how to improve the raytracer performance by using data-structures. Keywords-Ray Tracing; kD-Tree; Traversal; CUDA;

1 Introduction

Talk about raytracing and rendering and data structures

2 Object subdivision

3 Spatial Subdivision

4 Introduction

The aim of this lab is to build a simple ray tracer capable of rendering images using some of the concepts we learned in our computer graphics course. I would like to thank **Prof. Dr.-Ing. Matthias Teschner** for advising me throughout this lab. I would also like to mention some important resources I used to build the ray tracer including, **Peter Shirley's Ray tracing in One Weekend** (?, ?), **Kevin Suffern's Ray tracing from the Ground Up** (?, ?) and **Scratch a pixel** (?, ?) which is a blog comprising of explanations of different concepts about

ray tracing. Other than these resources, I have used a couple of other resources including **Wikipedia** for which I have given references where ever necessary.

5 Why Ray tracing?

One of the biggest motivation in the field of Computer Graphics is to create virtual environments that look as close to how they look in real life. **Ray tracing** is capable of producing highly realistic images that have various applications including but not limited to movies, simulation as well as video games (recently). Ray tracing, unlike **rasterization** is capable of simulating light transport because of all the interactions between lights and objects. Disney's **Hyperion Renderer** for example is a Path tracer (an efficient ray tracing technique) which helps produce their animated movies.
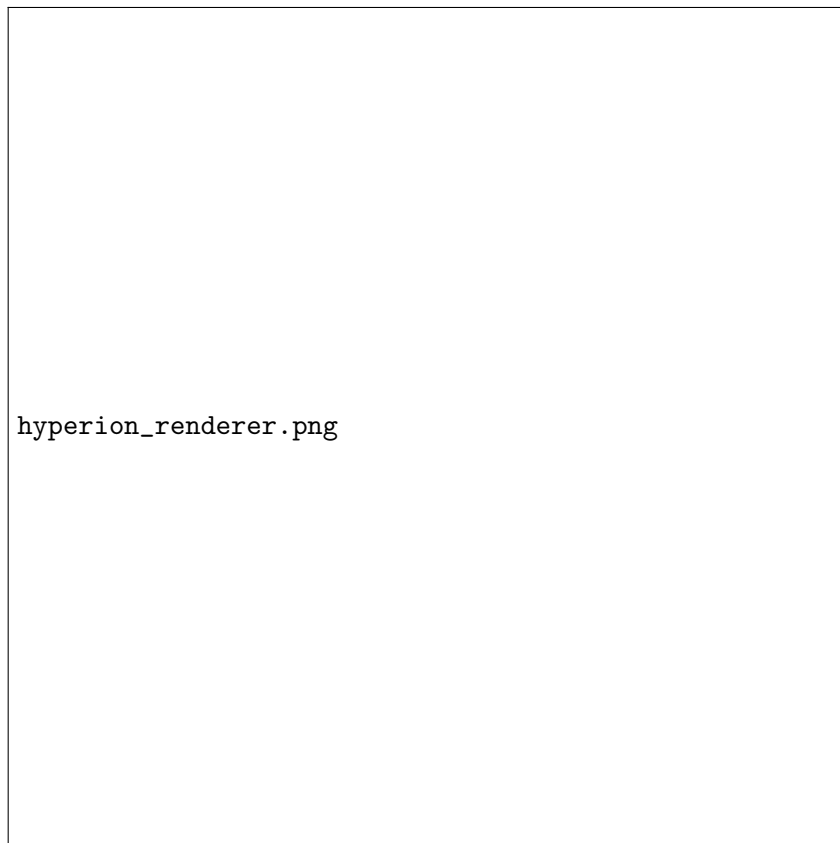

Figure 1: An image rendered by Disney's Hyperion Renderer (?, ?).

As you can observe in the image above, it is almost impossible to figure out if this is an actual photograph of real world objects or rendered by a computer. You can also notice how different objects are able to simulate different material properties similar to how they look in real life. Nowadays, the use of ray tracing is also becoming main stream in interactive applications such as video games because of the development of high performance GPUs. **NVIDIA Marbles** for example, is a fully playable physics based game which also makes use of recently introduced Nvidia's RTX (real-time ray tracing) technology.

nvidia_marbles.png

Figure 2: A snapshot of NVIDIA Marbles at Night Demo. (?, ?).

So these examples clearly demonstrate that ray tracing can be used in a variety of applications while also giving more realistic results. While it is still quite expensive as compared to rasterization it is becoming more and more mainstream as the computing potential in GPUs and CPUs improve from time.
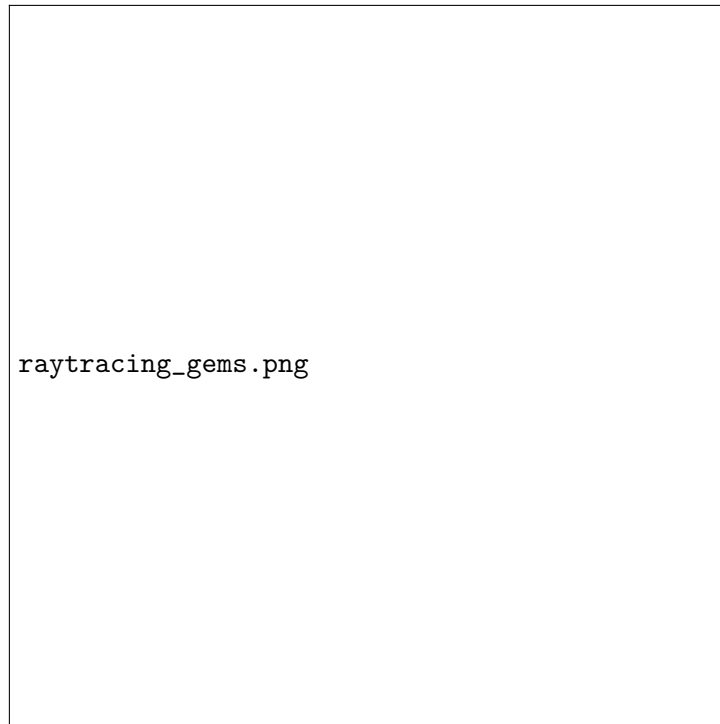
6 How ray tracing works?

Figure 3: An image showing how a ray is casted from the camera and how objects are illuminated and shadows are computed in ray tracing (?, ?).

Ray tracing makes use of **ray casting** to compute the closest hit point with an object. This is done by a camera that shoots multiple rays from each pixel of the image. These rays then intersect with an object in the scene which is evaluated by a ray-object intersection method. Each object in our scene also have different material properties that bounce off light differently. Shadows are computed by sending another ray from the hit point towards the light source and if this ray hits another object on its way towards the light source, then that hit point is in shadow. This process can be done recursively to get more realistic results keeping a recursion cut off limit. The aim of ray tracing is to compute the overall light transport in a scene while illuminating surfaces according to their material properties.

7 Implementation Details

The accompanied ray tracer uses C++ programming language to implement it. I worked on this ray tracer using many resources but the initial structure follows the one explained in **Peter Shirley's Ray tracing in One Weekend** (?, ?) therefore, some parts of the code are quite similar to that book. The code was implemented and tested on a machine with Ubuntu 20.04.2 LTS. Here, I will mention briefly the structure of the ray tracer.

7.1 External Libraries

The ray tracer uses two external libraries. Both of these libraries are single file header only, so nothing needs to be installed in order to make the code run.

1. **pnm:** This is a ppm image read/write library for C++. This is to facilitate in loading different kinds of textures as some ppm files usually have a binary encoding.

2. **tinyobjloader:** This is a light weight .obj format parser for C++. This is to facilitate in loading different .obj format meshes.

7.2 Code Structure

1. **camera.h, vec3.h, sphere.h, color.h, constants.h, hittable.h, ray.h**
   These header files are quite similar to how they were implemented in Ray tracing in One Weekend. **cam.h** implements a perspective camera. **vec3.h** implements the vector class along with different vector helper functions. **color.h** has a function that writes the color of the pixel to an image file. **constants.h** have some global helper functions and constants. **hittable.h** contains a struct that saves the hit point properties when a ray is hit with an object. **sphere.h** is the sphere class with its definitions as well as its intersect method. The intersect method is the same as the one being used in the book but it also adds functionality for textures in it. **ray.h** is the ray class.

2. **triangle.h, aabb.h, light.h, mesh.h, texture.h, scene.h**
   These header files are a mix of different resources I went through in order to implement them. This includes going through different books as well as articles. **triangle.h** is the triangle class with its definitions as well as the intersect method. The intersect method follows the one by (?, ?) which is the same algorithm I've explained in the report. **aabb.h** is axis aligned bounding boxes implementation and follows the one explained at (?, ?). **light.h** has two different light classes. The random sampling of positions in area lights follows an implementation discussed at (?, ?). **mesh.h** uses tinyobjloader to load a mesh. **texture.h** has a texture class that uses pnm to load a texture. **scene.h** is the biggest class in the ray tracer and implements phong illumination and implements how a hit point is shaded according to the type of light used. This also implements antialiasing as explained in Ray tracing in One Weekend (?, ?).

8 Using PPM (Portable Pixmap) format to generate images

PPM (Portable Pixmap) image format is used to generate images. One reason for using this format is that it is easier to generate images in this format as compared to other formats. It is based on RGB color space, red(R), green(G) and blue(B). The color component can be specified by a value between 0 and a maximum value of 255. A typical PPM image file looks like this:

```
P3
# P3 means colors are in ASCII, then 3 columns and 3 rows,
# then 255 for max color, then RGB triplets
3 3
255

255 0 0        0 255 0        0 0 0
0 0 255        255 255 0      255 0 255
255 255 255    0 255 255      255 100 0
```
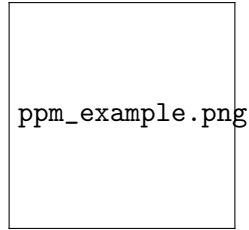
which generates an image:

Figure 4: A ppm image with the values as mentioned above. This example is magnified as these are individual pixel colors and might show up small on an actual image (?, ?).

9 Sending rays into the scene

A ray can be defined as a function $\boldsymbol{p}(t) = \boldsymbol{o} + t\boldsymbol{d}$ where $\boldsymbol{p}$ is a position along a line in 3D. $\boldsymbol{o}$ is the origin (the starting point of the ray) while $\boldsymbol{d}$ is the ray direction. t is how far we want the ray to travel in the direction $\boldsymbol{d}$.
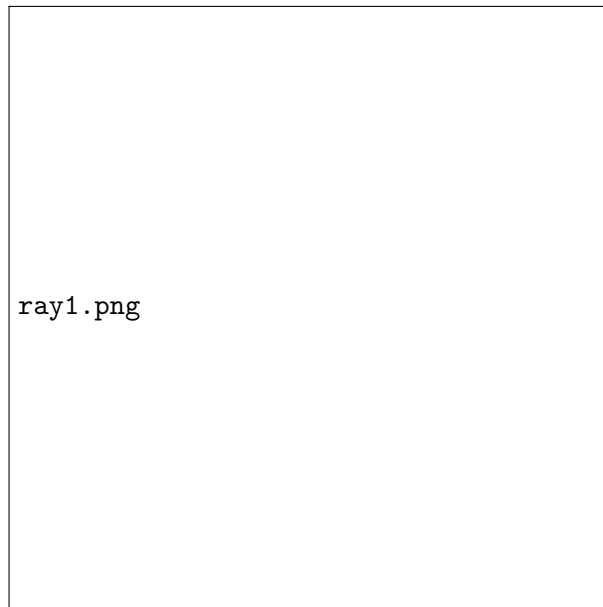
Figure 5: A ray with origin $\boldsymbol{o}$ at $t = 0$ going towards $t = 1$ with a direction $\boldsymbol{d}$ (?, ?).

There are 3 important steps for a ray tracer:

1. Calculate the primary ray from the eye (camera) to the pixel.

2. Determine the ray's closest intersection with an object.

3. Compute a color for that intersection point.

For setting up the camera we'll set the $y - axis$ to go up, the $x - axis$ to the right and the negative $z - axis$ into the screen as shown in the figure below:

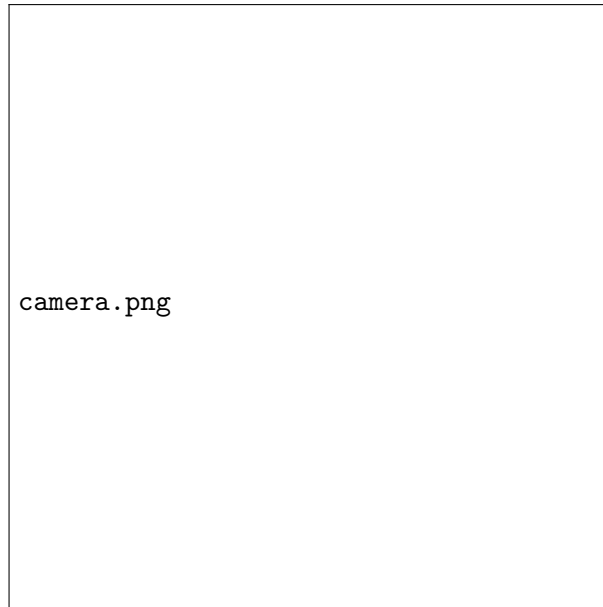Figure 6: A camera(eye) setup (?, ?).

10 Adding a triangle to the scene (ray-triangle intersection)

10.1 Introduction

In computer graphics, most 3D models are triangulated (you can make up any geometry combining a bunch of triangles into a triangular mesh) so starting with the most important shape is a good idea.

There are some properties related to vectors that are important here:

1. Subtracting a point a from a point b in space would give you a vector $\boldsymbol{a}$ along the direction of $\boldsymbol{b}$.

2. A cross product between two vectors, $\boldsymbol{a} \times \boldsymbol{b}$ gives a vector that is perpendicular to both of them. This is important in our case because this allows us to get the normal, $\boldsymbol{n}$ of the plane our triangle lies in.
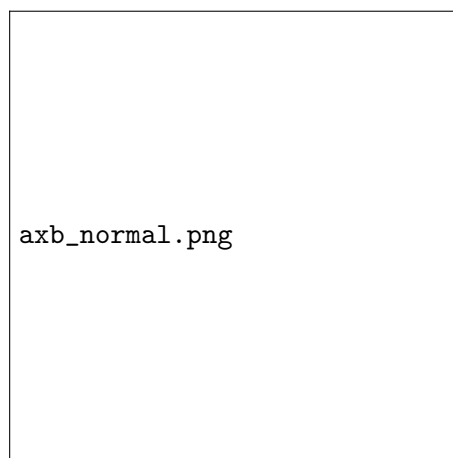

Figure 7: Vector $\boldsymbol{a}$ and $\boldsymbol{b}$ and the normal $\hat{\boldsymbol{n}}$.

3. A dot product between two vectors, $\boldsymbol{a}.\boldsymbol{b}$, if 0, means that the vectors are orthogonal to each other.

10.2 Barycentric Coordinates

We all know defining a triangle is easy. We can have three points in space and can define a triangle. But we need to somehow figure out the position of a given point $\boldsymbol{p}$ in space given three points of a triangle.
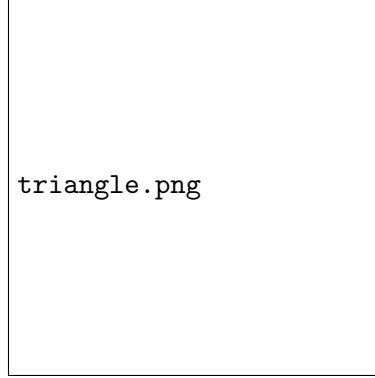

Figure 8: Three points, $\boldsymbol{p}_0$, $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ make up a triangle. A point $\boldsymbol{p}$ lies on that triangle (?, ?).

Because triangles are planar surface and the points that make up the triangle uniquely define that plane. Any point $\boldsymbol{p}$ on that plane, can be represented as a linear combination of the points that make up the triangle. So in our example above, our point $\boldsymbol{p}$ can be represented as a linear combination of the points, $\boldsymbol{p}_0$, $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$:

$$\boldsymbol{p} = \alpha\boldsymbol{p}_0 + \beta\boldsymbol{p}_1 + \gamma\boldsymbol{p}_2 \tag{1}$$

These α, β and γ are called the Barycentric Coordinates of this point $\boldsymbol{p}$ on this triangle. These are the weights (in simpler terms) that we could use to determine the position of point $\boldsymbol{p}$ using a linear combination of the vertices $\boldsymbol{p}_0$, $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$. These barycentric coordinates also have a very important property which is:

$$\alpha + \beta + \gamma = 1 \tag{2}$$

Note, that if we know any of the two barycentric coordinates, α and β, we can get the third one as $\gamma = 1 - \alpha - \beta$ as they must add upto 1. One important thing to mention is that we can represent any point on the infinite plane our triangle lies in, using these coordinates. This includes a point $\boldsymbol{p}$ which is outside our triangle. But if our point $\boldsymbol{p}$ is on the triangle, we have an additional property that our barycentric coordinates would always be positive and smaller than 1:

$$0 \leq \alpha \leq 1 \quad 0 \leq \beta \leq 1 \quad 0 \leq \gamma \leq 1 \tag{3}$$

One advantage of using barycentric coordinates is that they could be used to define attributes (color or texture) on a triangle because we are essentially linearly interpolating between the three vertices of a triangle.

10.3 Möller-Trumbore Algorithm

The **Möller-Trumbore Algorithm** is a fast ray-triangle intersection algorithm which was introduced by **Tomas Möller** and **Ben Trumbore** in 1997 in their paper titled "Fast, Minimum

Storage Ray/Triangle Intersection" (?, ?).

A point, $\boldsymbol{t}(\alpha, \beta)$, on a triangle can be represented as:

$$\boldsymbol{t}(\alpha, \beta) = (1 - \alpha - \beta)\boldsymbol{p}_0 + \alpha\boldsymbol{p}_1 + \beta\boldsymbol{p}_2 \qquad (4)$$

Here, (α, β) are the barycentric coordinates of a triangle which hold all the properties we mentioned above. The intersection between ray, $\boldsymbol{p}(t)$, and the triangle, $\boldsymbol{t}(\alpha, \beta)$, is equivalent so:

$$\boldsymbol{o} + t\boldsymbol{d} = (1 - \alpha - \beta)\boldsymbol{p}_0 + \alpha\boldsymbol{p}_1 + \beta\boldsymbol{p}_2 \qquad (5)$$

Expanding this equation, we get:

$$\boldsymbol{o} + t\boldsymbol{d} = \boldsymbol{p}_0 - \alpha\boldsymbol{p}_0 - \beta\boldsymbol{p}_0 + \alpha\boldsymbol{p}_1 + \beta\boldsymbol{p}_2 \qquad (6)$$

And rearranging we get:

$$\boldsymbol{o} - \boldsymbol{p}_0 = -t\boldsymbol{d} + \alpha(\boldsymbol{p}_1 - \boldsymbol{p}_0) + \beta(\boldsymbol{p}_2 - \boldsymbol{p}_0) \qquad (7)$$

We have three unknowns here t, α, β with four known terms $\boldsymbol{o}, (\boldsymbol{p}_1 - \boldsymbol{p}_0), (\boldsymbol{p}_2 - \boldsymbol{p}_0)$ and $\boldsymbol{d}$. We can again rearrange these terms into the form:

$$\begin{bmatrix} -\boldsymbol{d} & (\boldsymbol{p}_1 - \boldsymbol{p}_0) & (\boldsymbol{p}_2 - \boldsymbol{p}_0) \end{bmatrix} \begin{bmatrix} t \\ \alpha \\ \beta \end{bmatrix} = \boldsymbol{o} - \boldsymbol{p}_0$$

This means, that solving the above linear system of equations, we can get our three unknowns (t, α, β). You can notice, that the authors of the paper are representing the coordinates of the point $\boldsymbol{t}$ in terms of α and β (the barycentric coordinates) of the triangle and not in terms of x, y and z. We know that α and β both can't be lower than 0 and $\alpha + \beta \leq 1$ so our triangle is in fact a unit triangle.

We denote $\boldsymbol{e}_1 = (\boldsymbol{p}_1 - \boldsymbol{p}_0)$, $\boldsymbol{e}_2 = (\boldsymbol{p}_2 - \boldsymbol{p}_0)$ and $\boldsymbol{c} = \boldsymbol{o} - \boldsymbol{p}_0$, then by **Cramer's Rule** (?, ?) we can solve the above system of equations as:

$$\begin{bmatrix} t \\ \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \frac{1}{|-\boldsymbol{d}, \ \boldsymbol{e}_1, \ \boldsymbol{e}_2|} \end{bmatrix} \begin{bmatrix} |\ \boldsymbol{c}, \ \boldsymbol{e}_1, \ \boldsymbol{e}_2 \ | \\ |-\boldsymbol{d}, \ \boldsymbol{c}, \ \boldsymbol{e}_2| \\ |-\boldsymbol{d}, \ \boldsymbol{e}_1, \ \boldsymbol{c}| \end{bmatrix}$$

where $|-\boldsymbol{d}, \ \boldsymbol{c}, \ \boldsymbol{e}_2|$ is the determinant of the matrix having columns, $-\boldsymbol{d}, \boldsymbol{c}$ and $\boldsymbol{e}_2$. Finally, since we are dealing with vectors, we can use the property:

$$|\boldsymbol{u}, \ \boldsymbol{v}, \ \boldsymbol{w}| = -(\boldsymbol{u} \times \boldsymbol{w}) \cdot \boldsymbol{v} = -(\boldsymbol{w} \times \boldsymbol{v}) \cdot \boldsymbol{u} \qquad (8)$$

and write our solution as:

$$\begin{bmatrix} t \\ \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \frac{1}{(\boldsymbol{d} \times \boldsymbol{e}_2) \cdot \boldsymbol{e}_1} \end{bmatrix} \begin{bmatrix} (\boldsymbol{c} \times \boldsymbol{e}_1) \cdot \boldsymbol{e}_2 \\ (\boldsymbol{d} \times \boldsymbol{e}_2) \cdot \boldsymbol{c} \\ (\boldsymbol{c} \times \boldsymbol{e}_1) \cdot \boldsymbol{d} \end{bmatrix}$$

or in a more concise way:

$$\begin{bmatrix} t \\ \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \frac{1}{\boldsymbol{p} \cdot \boldsymbol{e}_1} \end{bmatrix} \begin{bmatrix} \boldsymbol{q} \cdot \boldsymbol{e}_2 \\ \boldsymbol{p} \cdot \boldsymbol{c} \\ \boldsymbol{q} \cdot \boldsymbol{d} \end{bmatrix}$$

where $\boldsymbol{p} = (\boldsymbol{d} \times \boldsymbol{e}_2)$ and $\boldsymbol{q} = (\boldsymbol{c} \times \boldsymbol{e}_1)$.

We compute t, α and β and return -1 (*no intersection*) if

$$\alpha < 0 \ or \ \alpha > 1 \ or \ \beta < 0 \ or \ (\alpha + \beta) > 1 \qquad (9)$$

otherwise, there's an intersection and we return t, α and β.
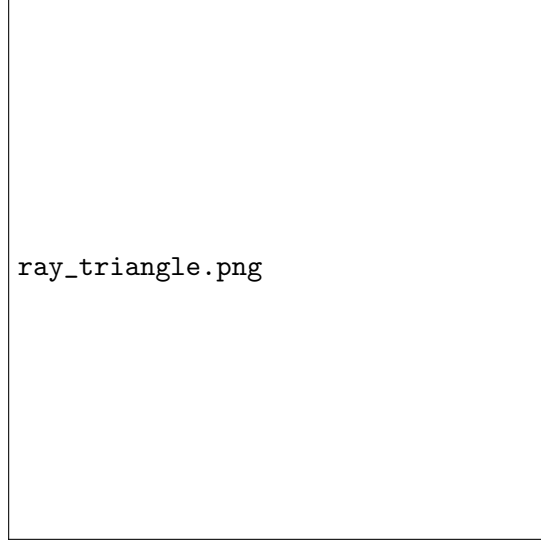
Figure 9: A triangle using normal as color value.

11 Adding a sphere to the scene (ray-sphere intersection)

Let's start with circle first in order to get some intuition. The equation of a circle centered at origin of radius r is $x^2 + y^2 = r^2$. If a given point (x, y) is inside the circle, then $x^2 + y^2 < r^2$ and if it is outside the circle, then $x^2 + y^2 > r^2$. Considering the center is at (c_x, c_y), our equation becomes:

$$(x - c_x)^2 + (y - c_y)^2 = r^2 \tag{10}$$

An example would make it more clear. Consider a circle with its origin at $(4, 4)$ and radius 2 along with three points, $\boldsymbol{p}_0(6, 4)$, $\boldsymbol{p}_1(5, 3)$ and $\boldsymbol{p}_2(6, 1)$. $\boldsymbol{p}_0$ is on the circle, $\boldsymbol{p}_1$ is inside while $\boldsymbol{p}_2$ is outside and we can verify this with our equation.
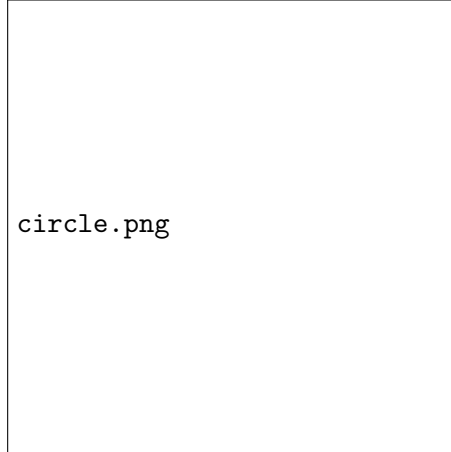

Figure 10: A circle with its origin at $(4, 4)$ along with 3 other points.

$$(6 - 4)^2 + (4 - 4)^2 = (2)^2 \tag{11}$$

$$(5 - 4)^2 + (3 - 4)^2 < (2)^2 \tag{12}$$

$$(6 - 4)^2 + (1 - 4)^2 > (2)^2 \tag{13}$$

But since we want to render a sphere and not a circle we'll be using it's equation which is the same as circle's equation but with added dimension z. So our equation for sphere with its center at (c_x, c_y, c_z) is:

$$(x - c_x)^2 + (y - c_y)^2 + (z - c_z)^2 = r^2 \tag{14}$$

And the same rules for points being on, inside or outside the circle applies for sphere too. Since we are dealing with vectors, we need to change equation (14) in vector form. Remember that a vector from a point $\boldsymbol{a}$ to point $\boldsymbol{b}$ is given by $\boldsymbol{b} - \boldsymbol{a}$, a vector from our center $\boldsymbol{c}$ to point $\boldsymbol{p}$ can similarly be given by $(\boldsymbol{p} - \boldsymbol{c})$ and equation (14) becomes:

$$(\boldsymbol{p} - \boldsymbol{c}) \cdot (\boldsymbol{p} - \boldsymbol{c}) = r^2 \tag{15}$$

Adding our ray equation, to equation (15) for all points t that satisfies the equation we get:

$$(\boldsymbol{o} + t\boldsymbol{d} - \boldsymbol{c}) \cdot (\boldsymbol{o} + t\boldsymbol{d} - \boldsymbol{c}) = r^2 \tag{16}$$

Expanding equation (16) and moving all terms to the left side we get:

$$t^2 \boldsymbol{d}^2 + 2t\boldsymbol{d} \cdot (\boldsymbol{o} - \boldsymbol{c}) + (\boldsymbol{o} - \boldsymbol{c}) \cdot (\boldsymbol{o} - \boldsymbol{c}) - r^2 = 0 \tag{17}$$

If you observe closely, equation (17) is a quadratic equation with an unknown t. We can get the discriminant using the formula we all have learned in our high school:

$$b^2 - 4a \cdot c \tag{18}$$

Our discriminant is zero when we have one real solution (**ray intersects the boundary of the sphere**), it is positive when we have two real solutions (**ray intersects two points, the one where it enters and the other where it leaves**) and it is negative when we have no real solutions (**the ray did not intersect**).
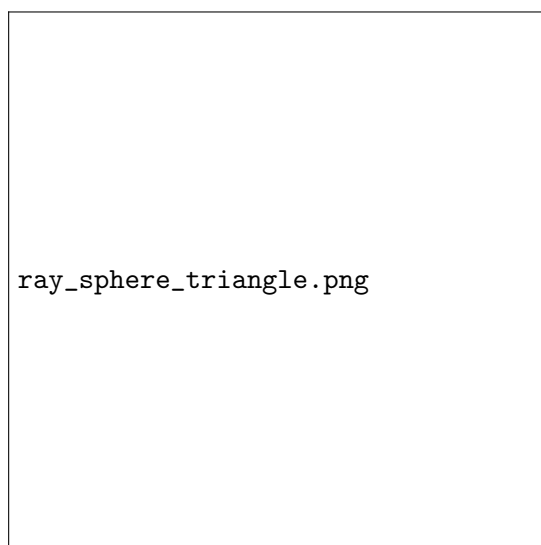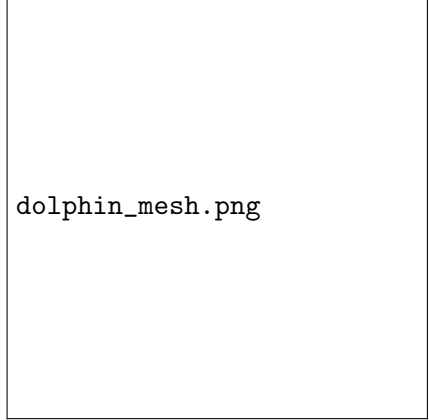

Figure 11: A sphere on a plane made up of two triangles using normal as color value.

12 Adding a triangular mesh to the scene

Since we can intersect triangles now with our ray-triangle intersection. We can use the same method to load **triangular meshes**. A triangular mesh is a type of polygon mesh which comprises of a bunch of triangles. Just like a cube can be formed using 12 triangles (2 triangles for each face of the cube), you can create any 3D model with a bunch of triangles and the detail of the model depends on the amount of triangles the model has. The more the number of triangles in a mesh, the more accurately it can represent a model. Figure 12 shows an example of a triangular mesh of a dolphin. In the case of our ray tracer what we want to do is load all the triangles our mesh is made of, intersect all of those triangles using our ray-triangle intersection method and then compute the color of the points on all the triangles our ray intersects with. This is however an expensive process since many triangular meshes can comprise of thousands or even millions of triangles. The famous **Stanford Bunny** (?, ?) for example consists of about 69, 451 triangles. There are many methods to speed up the process of intersecting with complex objects, one of which would be discussed later in this report. The **normal** of a triangle would also play an important role in the next section when we move on to shade our meshes. For now, we can try to color our triangles with a constant color using their normals.

dolphin_mesh.png

Figure 12: A triangular mesh representing a dolphin (?, ?).

normal_mesh.png

Figure 13: Stanford Bunny on a plane visualized with constant colors according to normals. The one used here has 4, 968 triangles.

13 Shading

Now that we can do ray-sphere and ray-triangle intersections, we can implement some kind of shading. Shading refers to the process that gives visual appearance to objects in 3D space. Without shading, as you have observed in the image we generated previously, we can only show

constant colors on our objects. But, what we really want to achieve is to show objects as close to how they look in real life. We would want glossy surfaces to look shiny while matte surfaces should look dull. What's needed to simulate materials of these kind, is some kind of **illumination**.

Illumination is a general term for light that originates at the light sources and arrives at object surfaces. Illumination can be divided into generally two types: Direct or Indirect. **Direct illumination** refers to the light that hits a surface directly from a light source while **indirect illumination** is the light that hits a surface after reflecting off from at least one surface. Direct illumination is also often referred to as **local illumination** while indirect illumination is referred to as **global illumination**. For the accompanied ray tracer, a well known local illumination model known as **Phong Illumination** model (?, ?) has been implemented. It can simulate light as well as material shading. But it is important to discuss lights first, which would help us illuminate surfaces.

A light source in our scene is a virtual object that lets us illuminate other objects in our scene. A **light source** is defined with a **color** as well as **intensity**.

13.1   Point light

I have implemented **Point light** in my ray tracer which is a type of spherical light that emits light in all directions similar to a light bulb. They also only emit light from a single point in space. A point light's intensity reaching a point p on an object or anything in our scene also depends on how far the light is from that object. In fact, light source attenuation follows the **inverse square law** (?, ?) which states that a quantity (in our case, light intensity at a point) is inversely proportional to the square of the distance from source of the quantity (a light source). Or in other words, light intensity at a point p decreases quadratically with the distance from the light position l_p.
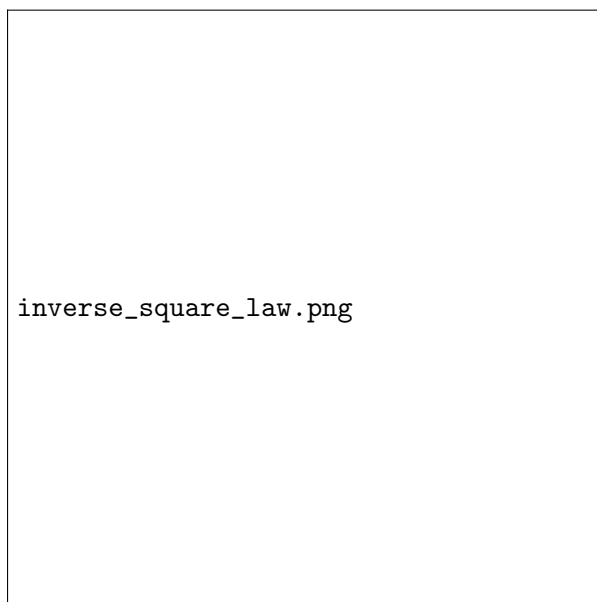
inverse_square_law.png

Figure 14: Light's area grows quadratically with distance r (?, ?).

For point lights, this can be formulated as:

$$l_{ip} = \frac{l_i}{4 * \pi * d^2} \tag{19}$$

where l_{ip} is the actual intensity reaching a point $\boldsymbol{p}$, l_i is the intensity value of the light and d is the distance between the light position $\boldsymbol{l}_p$ and point $\boldsymbol{p}$.

13.2   Diffuse Reflection

Diffuse reflection simulates matte like surfaces. Surfaces that don't have shiny appearance and that reflect light in many directions. Surface illumination strength can be calculated by the dot product of the surface normal $\boldsymbol{n}$ and light source's direction $\boldsymbol{l}_d$ which is referred to as the **Lambert's Cosine Law** (?, ?). In other words, illumination strength is directly proportional to the cosine of the angle between $\boldsymbol{n}$ and $\boldsymbol{l}_d$. If light intensity reaching a point $\boldsymbol{p}$ is given by l_{ip} (as calculated above) and $\boldsymbol{\rho}$ is the albedo (hit point color) we can calculate the diffuse reflection at a point as:

$$\boldsymbol{l}_{diffuse} = \boldsymbol{\rho} * \frac{1}{\pi} * (\boldsymbol{n} \cdot \boldsymbol{l}_d) * \boldsymbol{l}_{color} * l_{ip} \tag{20}$$

Where, $\boldsymbol{l}_{color}$ is the light's color. This can be done for multiple lights and the formula becomes:

$$\boldsymbol{l}_{diffuse} = \Sigma_i \boldsymbol{\rho} * \frac{1}{\pi} * (\boldsymbol{n} \cdot \boldsymbol{l}_{i_d}) * \boldsymbol{l}_{i_{color}} * l_{i_{ip}} \tag{21}$$

So the total diffuse reflection $\boldsymbol{l}_{diffuse}$ reaching a point $\boldsymbol{p}$ is the sum of all the diffuse reflections calculated from all lights in the scene.

diffuse_reflection.png

Figure 15: A blue sphere with a point light at top-left.

13.3   Ambient Reflection

Ambient Reflection can be described as indirect illumination from other surfaces. In the accompanied ray tracer, the albedo of the surface is being used as the ambient value. This can

also be multiplied with a constant k_{amb} to increase or decrease the ambient intensity. Overall, ambient reflection can be given as:

$$l_{ambient} = \boldsymbol{\rho} * k_{amb} \tag{22}$$

Combining our diffuse and ambient components, our formula for an individual point light becomes:

$$l_{total} = (\boldsymbol{\rho} * k_{amb}) + \boldsymbol{\rho} * \frac{1}{\pi} * (\boldsymbol{n} \cdot \boldsymbol{l}_d) * \boldsymbol{l}_{color} * l_{ip} \tag{23}$$

Figure 16: A blue sphere with a point light at top-left.

13.4   Specular Reflection

Specular Reflection simulates shiny surfaces. These kind of surfaces reflect light in some dominant directions instead of diffuse which reflects light in all directions. For producing specular highlights, we need a reflection vector $\boldsymbol{r}$ which is the reflected light from a shiny surface. This can be calculated as:

Figure 17: Reflection vector $\boldsymbol{r}$, view vector $\boldsymbol{v}$ and other useful vectors (?, ?).

$$r = 2 * (\boldsymbol{l} \cdot \boldsymbol{n}) * \boldsymbol{n} - \boldsymbol{l} \tag{24}$$

where vectors $\boldsymbol{l}$ and $\boldsymbol{n}$ are light direction and surface (hit) normal respectively. Both of these vectors should be normalized. After calculating the reflection vector, we need to calculate the view vector $\boldsymbol{v}$ which is the vector from the surface point $\boldsymbol{p}$ to the ray origin $\boldsymbol{r_o}$. After calculating both these vectors, the specular value of a surface point can then be calculated as:

$$l_{specular} = \boldsymbol{l}_{color} * (\boldsymbol{r} \cdot \boldsymbol{v})^m * k_{spec} \tag{25}$$

where m is sometimes called the **shininess** constant. We can get different specular highlights for different values of m. Overall reflected light depends on m. k_{spec} is just a constant to have the desired amount of specular reflection for a surface. Combining diffuse, ambient and specular gives us this formula for an individual point light:

$$l_{total} = (\boldsymbol{\rho} * k_{amb}) + \boldsymbol{\rho} * \frac{1}{\pi} * (\boldsymbol{n} \cdot \boldsymbol{l}_d) * \boldsymbol{l}_{color} * l_{ip} + \boldsymbol{l}_{color} * (\boldsymbol{r} \cdot \boldsymbol{v})^m * k_{spec} \tag{26}$$

Figure 18: A blue sphere with a point light at top-left. For this render, k_{amb} was 0.05, m was 10, k_{spec} was 0.1 and $\boldsymbol{l}_{color}$ was $(1, 1, 1)$ with l_{ip} of 30.

14   Shadows

For adding shadows in our scene, we need to compute an additional ray from the point of surface where our ray hits $\boldsymbol{p}$ towards the light position $\boldsymbol{l}_p$. If this ray which is often called as **shadow ray** hits another object on its way towards the light source, then the point that we are shading is in the shadow of that object.

We have to be careful though when we try to find out if a hit point $\boldsymbol{p}$ is in shadow of an

object and we only have to check the hit points which are between the point being shaded and the light location otherwise we can have an issue as shown below:
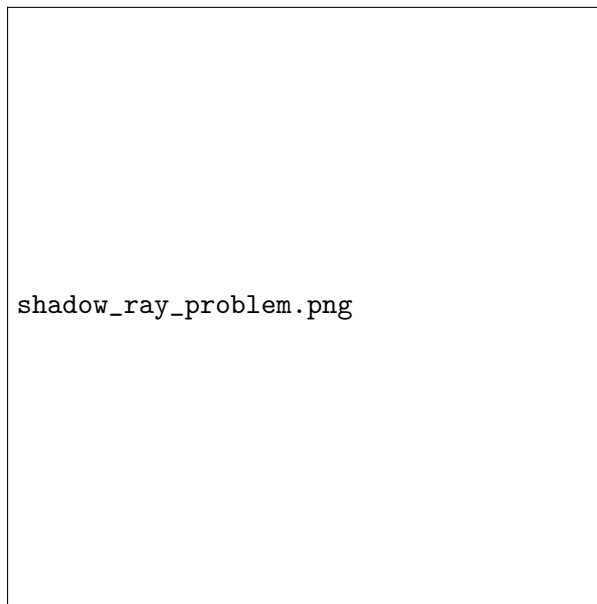

Figure 19: A point light between two spheres creates unexpected shadows on spheres.

Note the unexpected shadow that appears on both of these spheres. As we try to compute a shadow ray on a point on the yellow sphere, our ray (from hit point p towards l_p) goes beyond the light position and hits the pink sphere on the right side of the light and since it hit an object, we assume the point we were shading on the yellow sphere is in the shadow of an object while it actually isn't. The same happens with the pink sphere and we get an unexpected shadow. We can fix this by only testing our shadow ray between the position of our hit point p and the light position l_p after which we get the desired image as shown in figure 20.
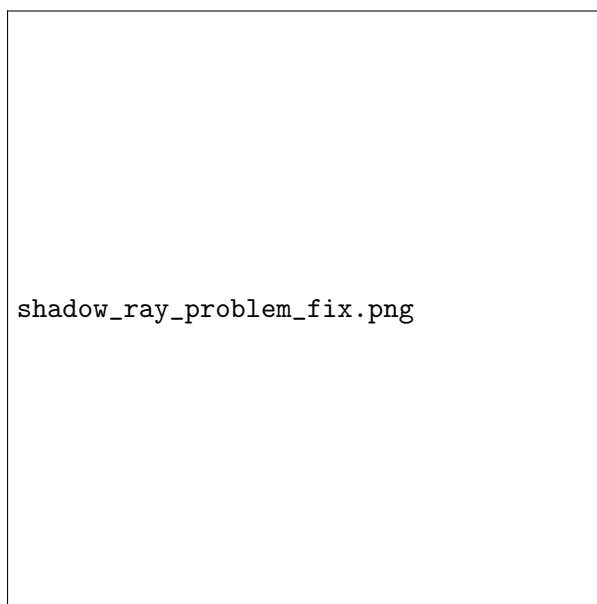

Figure 20: A point light between two spheres.

One another problem that often arises due to finite floating point precision is **shadow acne** or **salt and pepper noise**. The reason why that happens is due to the fact that when shadow rays intersect with an object, it is assumed that the hit function will return a value of 0 for t but that is not always the case. Sometimes, due to finite numerical precision, this t can be slightly positive ($3.245e - 07$) or slightly negative ($-4.567e - 08$) and these positive values are what causes the issue because then our hit function returns **true** and the shadow ray intersects the object where it was being cast. This can be fixed by introducing a small bias value and moving the shadow ray a bit towards the surface normal. The accompanied ray tracer uses a bias value of 0.001 so the shadow ray's origin becomes:

$$\boldsymbol{r}_o = \boldsymbol{p}_{hit} + \boldsymbol{n}_{hit} * 0.001 \tag{27}$$

where $\boldsymbol{p_{hit}}$ is the hit position and $\boldsymbol{n_{hit}}$ is the hit normal.

15   Antialiasing

So far the images we have produced have sharp jaggy edges. This is not so obvious if we look the image as a whole but still noticeable if the image is zoomed in a little.
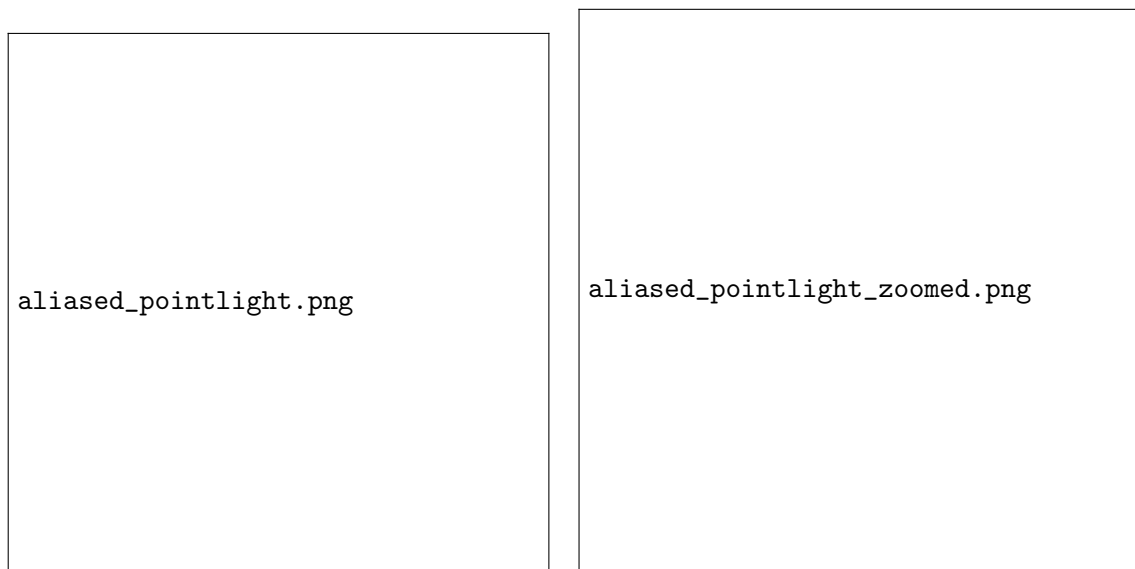

Figure 21: Image rendered has sharp and jaggy edges.

This is because we are only sending one ray per pixel from its center. Considering an image with width 6 pixels and height 5 pixels where a ray is sent to the center of the pixel can be shown as:
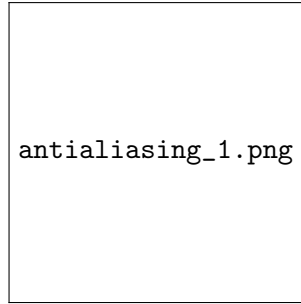
Figure 22: A 6x5 image where a ray is sent to the center of the pixel (?, ?).

But what we really want is an image like the one shown below, where the transition between colored pixels is smooth and not pixelated:
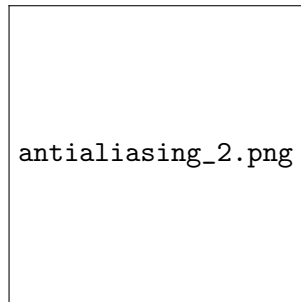

Figure 23: Desired image. (?, ?).

One way to achieve this is by sending multiple rays per pixel instead of just one. The ray tracer accompanied with this report sends multiple rays per pixel with a random range between $[0, 1)$ according to the desired amount of **anti aliasing** set. The color of all those rays are accumulated and then divided by the number of samples per pixel. So if we sent 10 rays per pixel, the returned colors of all those rays are added and then divided by 10. Note that for 10x anti aliasing, since we are sending 10 rays per pixel, this has also increased our time complexity by 10x so if we were sending 25 rays for a 5x5 image, now we are sending 250 rays for the same image.

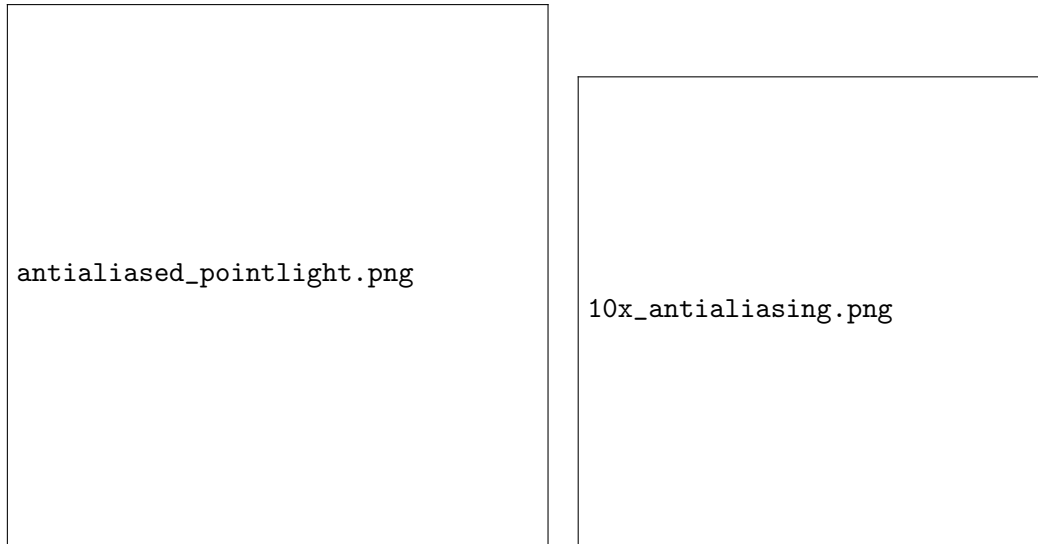Figure 24: Image rendered with 10x anti aliasing.

16   Texture mapping

Texture mapping is exactly what the term says. We have a texture, which can be any image of variable size and we want to map this image to a 3D object in our scene. An image's pixel is often represented as (u, v) coordinates that go from 0 to 1. If we have a grass texture for example, we can map the image to (u, v) coordinates as:

Figure 25: (u, v) coordinates being mapped from an image with resolution $h_{res} * v_{res}$ (?, ?).

Here, we have a texture image with horizontal resolution or width represented as h_{res} and vertical resolution or height represented as v_{res}. And the mapping from (u, v) to (x_p, y_p) can be given as:

$$x_p = (h_{res} - 1) * u \tag{28}$$

$$y_p = (v_{res} - 1) * v \tag{29}$$

16.1 Texturing background

Texturing our background is as simple as copying a pixel from the texture to our image that is being rendered. Since, we are going pixel by pixel as we render the image, we take the pixel's row and column number and divide it by width and height respectively, which gives us the (u, v) coordinate:

$$u = \frac{pixel_r}{image_w} \tag{30}$$

$$v = \frac{pixel_c}{image_h} \tag{31}$$

Where, $pixel_r$ is the pixel's row value, $pixel_c$ is the pixel's column value, $image_w$ is the width of the image being rendered and $image_h$ is the height of the image being rendered. So if we are rendering an image of width 1000 and height 562 and want to determine the texture value for pixel $(500, 286)$, we get (u, v) as $(0.5, 0.508)$. We can then use this (u, v) coordinate and equations (28) and (29) to get the pixel value from our texture.

16.2 Texturing a triangle

Texturing a triangle is also fairly easy. The algorithm we used for our ray-triangle intersection, computes t, α and β where α and β are actually two of the three **barycentric coordinates** of our triangle. These can be used to texture map our triangle, as barycentric coordinates essentially linearly interpolate between the three vertices of a triangle. Using α as u and β as v, we can use equation (28) and (29) to get the desired pixel from the texture and then use the pixel's (r, g, b) value as the albedo ($\boldsymbol{\rho}$) of our hit point on our triangle. Using the grass texture on two triangles making a plane gives us an image:

textured_triangles.png

Figure 26: Two textured triangles making a plane, with three point lights illuminating the scene. A stars texture is used as a background.

Note: Textures for meshes were not implemented due to time constraints.

16.3 Texturing a sphere

In order to texture map a sphere, we need to compute the (u, v) coordinates since we are not calculating them as part of our ray-sphere intersection algorithm. Just to review, our ray-sphere intersection returns a root t which is the scalar we use to get the hit position on our sphere using:

$$\boldsymbol{p_{hit}} = \boldsymbol{o} + t * \boldsymbol{d} \qquad (32)$$

where $\boldsymbol{o}$ is ray origin and $\boldsymbol{d}$ is the ray direction. After getting the hit position $\boldsymbol{p_{hit}}$ we get the normal on our sphere by:

$$\boldsymbol{n} = \frac{\boldsymbol{p_{hit}} - \boldsymbol{c}}{r} \qquad (33)$$
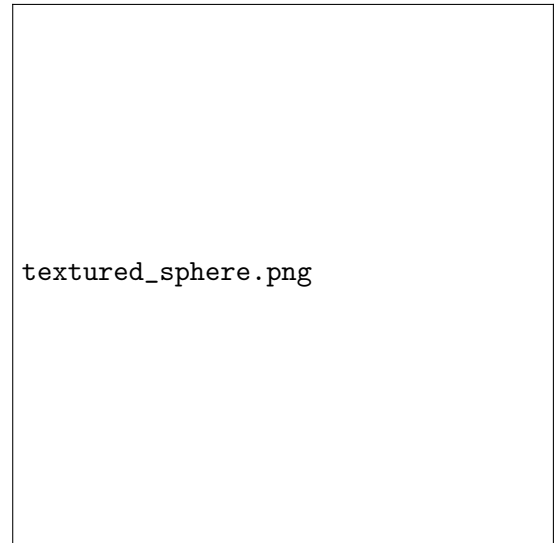
textured_sphere.png

Figure 27: Two textured triangles making a plane, a sphere with an earth texture in the middle with three point lights illuminating the scene. A stars texture is used as a background.

21

where c is the center of our sphere and r is the radius. A sphere has smooth normals on every point which means that n would be different for every p_{hit}. Once we have computed n, we can get the (u,v) coordinates for the sphere that would help us map a texture to our sphere:

$$u = 0.5 + \frac{arctan2(n_x, n_z)}{2 * \pi} \tag{34}$$

$$v = 0.5 - \frac{arcsin(n_y)}{\pi} \tag{35}$$

where n_x, n_y, n_z are the x, y and z coordinates of the normal n. Using equations (34) and (35) for (u,v) coordinates of our sphere, we can then use equations (28) and (29) to get the desired pixel from our texture.

17 Transformations

It would be convenient to translate, scale and rotate objects in our scene. In order to create complex scenes with lots of geometries, scattered all over our scene, it is useful to have some kind of transformation operations on our objects so that we can translate, scale or rotate them according to our needs. This is often done by the help of **affine transformations** which we will discuss one by one.

17.1 Translation

We will discuss **translation** first. This helps us moving objects in our scene to a particular position. This can be illustrated in 2D using vectors as shown in figure 28. In the figure, we have a vector p which is translated using a vector t resulting in a vector p'. This is essentially vector addition and can be given by:

Figure 28: A vector p is translated using a vector t resulting in a vector p' (?, ?).

$$p' = p + t \tag{36}$$

Note that for 3D, we must have an added dimension z in our vector. The above figure was just to make it simple to illustrate. In the case of a sphere, we will be applying this vector addition to its center, c and in the case of a triangle we will be doing this operation to each of its vertices v_0, v_1 and v_2. For meshes, this has to be done for each of the triangles' vertices in a mesh.

17.2 Scale

We would also like to **scale** objects in our scene. This helps changing the size of our objects, whether they are spheres or meshes. Each mesh also has a fixed size in their original format (.obj for example) and without scaling we won't be able to change its size. Scaling can be illustrated in 2D using vectors as shown in figure 29. We have a vector p which have been scaled to a vector p'. Note that the vector

Figure 29: A vector p is scaled to a resulting vector p' (?, ?).

is scaled in both x and y direction with the same amount which is also called **uniform scaling**. So essentially scaling is multiplication of a vector with a scaling factor s which can be given by:

$$p' = p * s \tag{37}$$

We can also do **non uniform scaling** by using a different scaling factor for each of the x, y and z coordinates of our vector. Which means that we want to scale our dimensions non uniformly. This can be given as:

$$p'_x = p_x * s_x \tag{38}$$

$$p'_y = p_y * s_y \tag{39}$$

$$p'_z = p_z * s_z \tag{40}$$

In the case of a sphere, scaling is multiplication of a scaling factor s with its radius r and in the case of a triangle it is the multiplication of a scaling factor with each of its vertices.

17.3   Rotation

Rotation is also useful since there will always be a need to rotate objects in our scene. Rotation in 2D using vectors can be illustrated as:
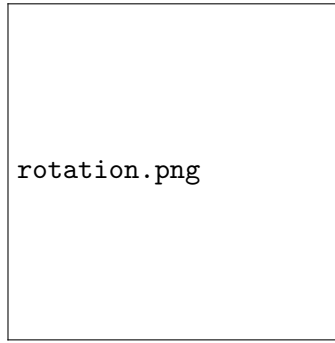

Figure 30: A vector $\boldsymbol{p}$ is rotated by an angle θ resulting in a vector $\boldsymbol{p'}$ (?, ?).

In 2D you can only rotate in clockwise or anti-clockwise direction. This can be thought of placing a paper on a table and then trying to rotate it without lifting it. And you quickly notice that it can only be done in clockwise or anti-clockwise manner. Anti-clockwise rotation in 2D which is the same shown in figure 30 can be given by:

$$p'_x = p_x * (cos\theta) + p_y * (-sin\theta) \tag{41}$$

$$p'_y = p_x * (sin\theta) + p_y * (cos\theta) \tag{42}$$

For clockwise rotation this becomes:

$$p'_x = p_x * (cos\theta) + p_y * (sin\theta) \tag{43}$$

$$p'_y = p_x * (-sin\theta) + p_y * (cos\theta) \tag{44}$$

But since, we are working in 3D, we need to be able to rotate at any axis. Rotation in 3D can be done in x, y or z axis or can also be a combination of rotation around these axis. Rotation around x axis is given by:

$$p'_x = p_x \tag{45}$$

$$p'_y = p_y * (cos\theta) + p_z * (sin\theta) \tag{46}$$

$$p'_z = p_y * (-sin\theta) + p_z * (cos\theta) \tag{47}$$

Rotation around y axis is given by:

$$p'_x = p_x * (cos\theta) + p_z * (-sin\theta) \tag{48}$$

$$p'_y = p_y \tag{49}$$

$$p'_z = p_x * (sin\theta) + p_z * (cos\theta) \tag{50}$$

And rotation around z axis is given by:

$$p'_x = p_x * (cos\theta) + p_y * (sin\theta) \tag{51}$$

$$p'_y = p_x * (-sin\theta) + p_y * (cos\theta) \tag{52}$$

$$p'_z = p_z \tag{53}$$

18 Camera

The **camera** in a ray tracer is the point where the rays start and go through each pixel. The accompanied ray tracer uses a virtual pinhole camera which implements perspective viewing. A perspective view, shows a scene the way it should look when a human sees it. Objects that are far should appear smaller and objects that are closer to the camera should appear bigger. This type of camera also gives more information about depth. The setting that is used in the ray tracer can best be visualized by figure 31. For a perspective view, all the rays have the same origin, which is at the viewpoint e. The directions however, are different for each pixel. The image plane is at some distance d in front of e. This distance is often also known as the **focal length** of a camera. It is important to mention **field of view (fov)** of the camera here. This is the extent of the virtual scene observed through the camera.

Figure 31: A visualization of a perspective view. The rays start at the viewpoint e and go through each of the pixels in an image. (?, ?).
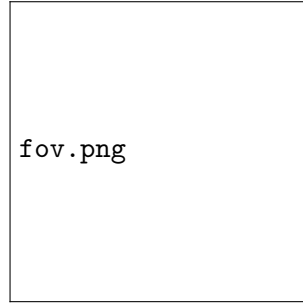

Figure 32: Field of view of a camera (?, ?).

As we move our camera, forwards or backwards (changing our z-axis), we need to be careful that the field of view is used to calculate a scaling factor for our view-port height and width. This scaling factor h is calculated as:

$$h = tan\left(\frac{\theta}{2}\right) \tag{54}$$

where θ is our field of view angle specified in radians. Our view-port width and height can then be calculated as:

$$view_h = 2 * h \tag{55}$$

$$view_w = aspect_r * view_h \tag{56}$$

where $aspect_r$ is the aspect ratio of the image and is typically kept at 16:9. We also need a way to specify a position to look from (e) so that we can view our scene from any point and a position to look at (l) so that the focus of our camera is on a specific point in a scene. This is achieved using an orthonormal basis (ONB) represented as (u, v, w) which can also be seen in figure 31 above. The vector w of the ONB is calculated as:

$$w = \frac{e - l}{||e - l||} \tag{57}$$

where e is the position we are looking from and l is the position we are looking at. The second vector u is calculated as:

$$u = \frac{up \times w}{||up \times w||} \tag{58}$$

where $\boldsymbol{up}$ is the up-vector usually taken as $(0, 1, 0)$ i.e y-axis. The third vector $\boldsymbol{v}$ is calculated as:

$$\boldsymbol{v} = \boldsymbol{w} \times \boldsymbol{u} \tag{59}$$

Figure 33: Three images rendered using the same scene placing camera at different positions. Three point lights of different colors were used to illuminate the scene.

19 Axis Aligned Bounding Box (Ray-AABB intersection)

So far, we have been using the ray-intersection methods we have implemented to check whether a ray hits an object. This means that for every ray that shoots from the camera, we are evaluating our spheres and triangles in our scene by going through their respective ray-intersection methods which can be quite expensive. It would be useful if we could wrap the objects in our scene with a simpler and less expensive ray-intersection method and when we are sure that the method returns true, we can then use the respective ray-intersection method for the actual object (sphere or triangle).

Figure 34: An axis-aligned bounding box (AABB) enclosing another object (?, ?).

This also speeds up the rendering process. It is quite obvious that if a ray does not return true for the less expensive intersection method, we can return from our hit function altogether, without even going through the expensive intersection method for the respective object. One way to achieve this is by using an **axis-aligned bounding box**. An axis-aligned bounding box completely encloses another object as illustrated in figure 34. It is called axis-aligned because it's axes are parallel to world coordinate axes. There are many methods introduced for ray-AABB intersections but the most dominant method is the **slabs test** (?, ?). A box requires two min and max positions which are the two opposite corners of the box. In 2D, it is represented by two slabs, an x-slab and a y-slab as shown in figure 35.

Figure 35: Left: A 2D axis-aligned box defined by two slabs. Right: A ray misses the box with non-overlapping x and y intervals (?, ?).

For 3D, we would have 3 slabs respectively. In the above figure, $\boldsymbol{p}_0$ and $\boldsymbol{p}_1$ are the min and max values. Figure 35 (Right) illustrates an example when a ray misses the box. The black arrow indicates the ray and the green dots indicate where the ray passes through the x-slab and the red dots indicate where the ray passes through the y-slab. The green and red dots create intervals on the ray. These are the intervals when the ray entered and exited the respective slabs. The ray intersects the box if these two intervals overlap which is clearly not the case above. Figure 36 illustrates some cases when a ray intersects the box and it can be seen that the two intervals also overlap.

Figure 36: Cases where a ray intersects the box (?, ?).

For a sphere, the min and max positions are given by:

$$\boldsymbol{min}_{aabb} = \boldsymbol{c} - r \tag{60}$$

$$max_{aabb} = c + r \tag{61}$$

where c is the center of the sphere while r is the radius. So radius r is subtracted from each x, y and z coordinate of the center c to get min_{aabb} while the radius is added to get max_{aabb}. For a triangle, min_{aabb} is the vertex with the minimum value of the x, y and z coordinate among all vertices and max_{aabb} is the one with the maximum value. For a mesh, this needs to be checked for all triangles in the mesh and the vertex with the minimum and maximum value is chosen.

19.1 Analysis of AABB

Now that we have an AABB implementation we can analyze the difference it makes when using AABBs as a first intersection test to encapsulate meshes in our scenes. When a mesh is loaded in the ray tracer, we will compute its min and max vertices as discussed in the previous section and create an axis-aligned bounding box for it. For every ray that goes into the scene from the camera's origin, if the ray doesn't hit the AABB of the mesh, it also would not hit the mesh itself and so we return from our hit function without doing any expensive computation (ray-triangle intersections). We will use three different scenes for this purpose and evaluate them by first rendering the scene without AABBs and then afterwards rendering the same scene using AABBs. Each of the three scenes were using 5x anti-aliasing, has a plane made up of two triangles with three point lights illuminating the scene:

1. The first scene uses a cube as a mesh made up of 12 triangles.

2. The second scene uses the stanford bunny (?, ?) as a mesh made up of 4, 968 triangles.

3. The third scene uses the utah teapot (?, ?) as a mesh made up of 6, 320 triangles.

We will calculate the number of **ray-mesh intersection** tests which is the number of times it is checked if a ray hits the mesh. We will also be calculating the number of **ray-triangle intersection** tests which is the number of times it is checked if a ray hits any of the triangles in the mesh. The counter for ray-triangle intersection tests does not take into account the checks for the two triangles that make up the plane as we are not using AABBs to enclose individual triangles. Note that the render times might vary on the machine used to render. These were the results I got on my machine.

19.1.1 Scene 1 - Cube

Figure 37: Image rendered of a slightly rotated cube.

Scene 1 - Cube			
Type	Ray-mesh intersection tests	Ray-triangle intersection tests	Time taken to render
Without AABB	6, 945, 413	80, 712, 814	11.16 seconds
With AABB	1, 186, 019	11, 600, 086	4.98 seconds

Scene 2 - Stanford Bunny			
Type	Ray-mesh intersection tests	Ray-triangle intersection tests	Time taken to render
Without AABB	7, 017, 974	30, 753, 145, 292	3, 665.91 seconds
With AABB	3, 280, 692	12, 186, 328, 316	1, 432.76 seconds

19.1.2   Scene 2 - Stanford Bunny

Figure 38: Image rendered of a stanford bunny.

19.1.3   Scene 3 - Utah Teapot

Figure 39: Image rendered of a utah teapot.

Scene 3 - Utah Teapot			
Type	Ray-mesh intersection tests	Ray-triangle intersection tests	Time taken to render
Without AABB	6, 987, 782	40, 894, 443, 281	3, 817.24 seconds
With AABB	3, 367, 474	18, 014, 096, 721	1, 623.18 seconds

It is quite obvious that using AABBs greatly reduces the time it takes to render an image. For the scenes tested, it can be seen that the time was halved from when the scenes were rendered without using AABBs. The intersection tests were also greatly reduced when using AABBs.

20   Area lights

This section is inspired by the online chapter on **Area Lights** from the book **The Ray Tracer Challenge** (?, ?). If you observe closely for the images rendered so far, it would be clear that the shadows generated using point lights are hard shadows with sharp edges.

Figure 40: Image rendered using three different point lights illuminating the scene. Note the hard shadows created by point lights.

A sphere would give an exact shadow of the shape of a sphere while this is not true when we observe shadows in real life. All light sources in real life such as a light bulb or a tube light have physical dimensions, which means that a point on an object can only be partially visible from these light sources. This partial occlusion, creates a **penumbra**, which is the part of a shadow where the light source is only partially blocked (?, ?). This results in more realistic looking shadows known as **soft shadows**.

Figure 41: Soft shadows (?, ?).

Soft shadows can be achieved using a different type of light source known as **area light**. Area lights, unlike point lights, are not a single point of light source but can be think of as a bunch

of light sources around a given position $\boldsymbol{l}_p$. Since we already have a point light implementation, we can use the same by using a number of point lights scattered around a given position. The current setting can be visualized as:

Figure 42: Point lights arranged in a 3x3 grid around a position $\boldsymbol{l}_p$.

Note that the above figure is only a 2D representation of how the lights are scattered while we're working in 3D. Let's try rendering the above setting.

Figure 43: Image rendered using the setting discussed above. 9 (3x3) point lights of white color were used to illuminate the scene.

A lot of things went wrong here. First, the sphere looks overexposed to light resulting in an incorrect shading. Second, while the shadows look different, they appear as if a number of shadows were combined on top of each other. We will tackle both of these problems one by one.

First, let's look at why the objects in our scene look overexposed to light. Since, we were using point lights, each of them had an intensity value l_i. The above rendered image had an intensity value of 30. And since we had 9 point lights scattered around a position $\boldsymbol{l}_p$, each of them had an intensity value of 30 and what we got was a combined effect in our image. What we really want to do is scale that intensity value according to the number of point lights we have in an area light. So, our intensity value should be scaled as:

$$l_i = \frac{l_i}{l_n} \tag{62}$$

Where l_i is the intensity value of a given point light and l_n is the total number of point lights in our scene (9 in our setting above). Let's render the scene again using the scaling we discussed.

Figure 44: Left: Image rendered again with scaled intensity. Right: Incorrect shadows being rendered.

This looks better but we still need to fix the incorrect shadows. This phenomenon is often called as **banding** where we get a series of stripes (bands) similar to what is happening with our shadows right now. This is happening in our case because we have point lights equally spaced from one another. What we can do to overcome this issue is, sample the positions of our point lights randomly.

Figure 45: Point lights arranged in a 3x3 grid but moved slightly away from their default position.

More simply, we move our point lights slightly away from their default position, using a radius r that defines how much we want to move our point lights away from their default position. This can be written as:

$$\boldsymbol{l}_p = \boldsymbol{l}_p + r * (2 * \eta - 1) \tag{63}$$

In the equations above, η is a random decimal value in the range of $[0, 1)$ and l_p is the default position of an individual point light. Rendering an image again after adding random sampling of positions gives us this image:

Figure 46: Image rendered after adding random sampling of light positions l_p.

While, it is not clear in the image above, random sampling also results in noisy images. This is noticeable when we reduce our point lights to a 2x2 grid and increase the light intensity to 60. One motivation to use a smaller grid is to reduce the time it takes to render our image as now we have a total of 4 point lights instead of 9 (3x3 grid) previously.

Figure 47: Image rendered using a 2x2 grid of point lights with a light intensity of 60.

Unfortunately there is no perfect solution to this issue. We can increase the grid size which tends to reduce the noise but then the image takes more time to render so it is always a trade off between time and quality. There are many papers that tend to reduce noise or the amount of time it takes to render images when using area lights and soft shadows but we will not be discussing them here. Let us now render the same image we rendered (figure 40) at the start of this section using area lights.

Figure 48: Image rendered using three different area lights illuminating the scene. Note the soft shadows created by area lights.

21   Final remarks

This lab was a great way to gain practical experience of the concepts we learned in our computer graphics course. I would like to thank **Prof. Dr.-Ing. Matthias Teschner** again for advising me throughout this lab. As a final remark, I would like to mention that I really enjoyed working on area lights. This includes debugging issues as I was working on them as well as how they were fixed. At the end, the results were worth the effort.

References