Implementation of a Raytracer in C++ Computer Graphics Lab (Rendering Track)

Alhajras Algdairy
alhajras.algdairy@gmail.com, 4963555, aa382

July 31, 2021

University:	University of Freiburg
Instructor:	Dr.-Ing. Matthias Teschner

Contents

1 Introduction

Computer graphics has three main pillars: *Modelling*, *Rendering*, and *Simulation*. The focus of this report is Rendering. Rendering cares about solving issues like the model's visibility towards sensors; this is done by two methods: *Rasterization* or *Raytracing*, this report is about a Raytracer implementation. Rendering also focuses on which color and intensity does a model has. How light interacts with surfaces and how it propagates throw the media.
The report summarizes the implementation of a simple Raytracer; this includes solving the Visibility challenge, coloring or shading the models and introducing some more features that enhance the Raytracer usability and performance.

Raytracing is a rendering technique that may provide extremely lifelike lighting effects. In other words, an algorithm can track the source of light and then mimic how the light interacts with the virtual objects it eventually encounters in the computer-generated environment. Raytracing produces far more lifelike shadows and reflections, as well as significantly enhanced translucence and dispersion. The algorithm considers where the light falls and calculates the interaction and interplay in the same way as the human eye does with actual light, shadows, and reflections. The way light interacts with items in the world has an impact on the colors you perceive.

Raytracer involves in different industries and applications. The media uses it because Raytracing can be beneficial in theater and television lighting since it allows for visually accurate simulation of light. Engineers also use the technique to forecast light levels, brightness gradients, and visual performance requirements in the following fields: Simulations, Modelling tools, and architectures.

To render a scene by the Raytracer, three crucial steps must be followed: *Casting Rays*: cast a ray for each pixel in the image. *Path tracing*: check if a ray crosses any of the scene's objects using the ray-geometry intersection, and trace the ray back to the source. *Shading*: determine how the object looks (color and brightness) at the point where the ray intersects the object.

Raytracing has two main methods:

- *Forward Raytracing*: The light particles (photons) are tracked from the light source to the object via Forward Raytracing. While forward ray tracing is the most exact method for determining the color of each object, it is also the most inefficient.

- *Backward Raytracing*: An eye ray is formed at the eye in backward ray tracing, and it travels through the viewplane and out into the scene. If it hits an object, it will return it to the viewplane immediately. This method is more efficient than Forward Raytracing but less accurate due to reducing the rays used. In this implementation this method is used.

2 Visibility

2.1 Introduction

The goal of rendering in computer graphics is to simulate light propagation to create images of virtual scenes. *Visibility* is a critical phenomenon that occurs as a result of light's interaction with the environment.
A fundamental graphics problem is determining the visible areas of surfaces and which object is closer to the camera. This challenge is referred to as visible surface determination or concealed surface removal, depending on how it is approached. The primary two operations to solve the Visibility challenge are: *Rays casting*, which shoots rays toward the scene, and *Path tracing* that traces the path of the shooted rays.

2.2 Rays

To generate an image using Raytracing, we must first cast a ray for each pixel in the image to the scene. Raytracing is a computer approach for simulating the behavior of light in a three-dimensional scene. It works by replicating genuine light rays and tracing the path that a beam of light would travel in the real environment using an algorithm. For this, all raytracers use rays as a way to simulate photons. Let us think of a ray as a function; here $\boldsymbol{r}$ is a 3D position along a line in 3D. $\boldsymbol{o}$ is the ray origin and $\boldsymbol{d}$ is the ray direction.

$$\boldsymbol{r}(t) = \boldsymbol{o} + t\boldsymbol{d}, \quad 0 < t < \infty \tag{1}$$

Figure 1 shows the basic idea of raytracing, where several rays are shooted toward the scene, and then a path tracer algorithm is used to shade the object and detect which parts are visible to the camera.

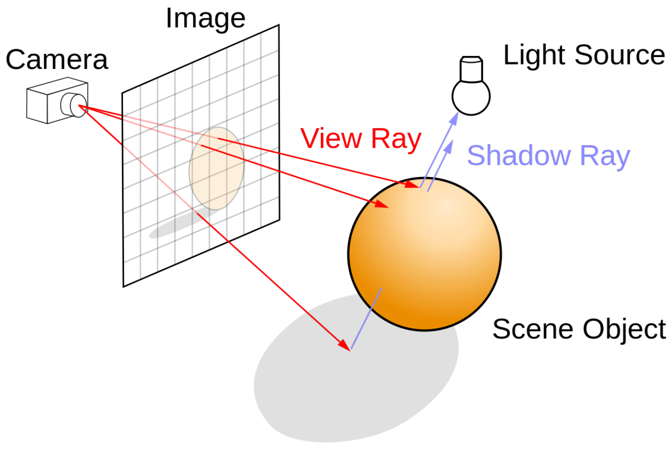


Figure 1: The raytracing algorithm builds an image by extending rays into a scene and bouncing them off surfaces and towards sources of light to approximate the color value of pixels [Piotr Dubla, "Interactive Global Illumination on the CPU."]

2.3 Rendering Sphere

For testing, spheres are often used in ray tracers because calculating whether a ray hits a sphere is pretty straightforward.
The general equation of a sphere with radius = 1 is:

$$(o_x + td_x)^2 + (o_y + td_y)^2 + (o_z + td_z)^2 = 1 \tag{2}$$

For solving the equation we need to use the Quadratic equation in t:

$$
\begin{aligned}
A(t)^2 + Bt + C &= 0 \\
A &= d_x^2 + d_y^2 + d_z^2 \\
B &= 2(d_x o_x + d_y o_y + d_z o_z) \\
C &= o_x^2 + o_y^2 + o_z^2 - 1 \\
t_{1,2} &= \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}
\end{aligned} \tag{3}
$$

By solving the equation we get three different cases as shown in Figure 2:

- No Intersection if: $B^2 - 4AC < 0$
- Single point of intersection if: $B^2 - 4AC = 0$
- Otherwise we get two points of intersection

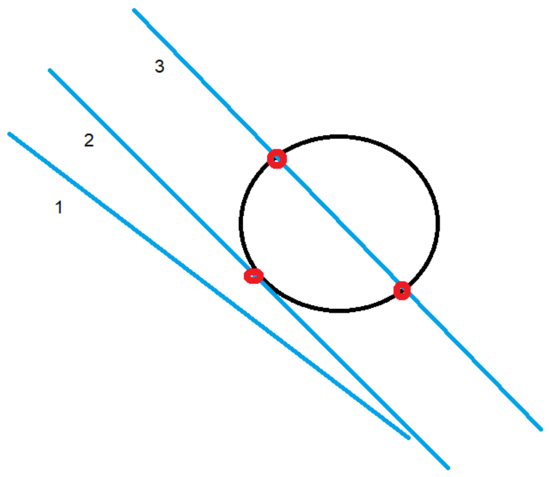


Figure 2: The three possible line-sphere intersections: 1. No intersection. 2. Single point intersection. 3. Two point intersection.

2.4 Rendering Triangle

Computing the intersection of a ray with a primitive such as a sphere is not tricky. However, because modeling most 3D objects with spheres alone is difficult, other types of primitives must be used to represent more complicated objects (objects of arbitrary shape). We may transform every object into a triangular mesh and compute the intersection of a ray with every triangle in this mesh instead of working with complex primitives like NURBS or Bezier patches. To represent a triangle a Parametric representation can be used, Parametric representation:

$$
\boldsymbol{p}(b_1, b_2) = (1 - b_1 - b_2)\boldsymbol{p}_0 + b_1\boldsymbol{p}_1 + b_2\boldsymbol{p}_2 \tag{4}
$$

Vertices $\boldsymbol{p}_0$, $\boldsymbol{p}_1$, $\boldsymbol{p}_2$ form a triangle. $\boldsymbol{p}$ is an arbitrary point in the plane of the triangle.
Figure 3 shows the three different cases that the ray can have with a triangle: Ray misses, Ray parallel and Ray intersects, we get an intersection if: $b_0 \geq 0 \wedge b_1 \geq 0 \wedge b_2 \geq 0$

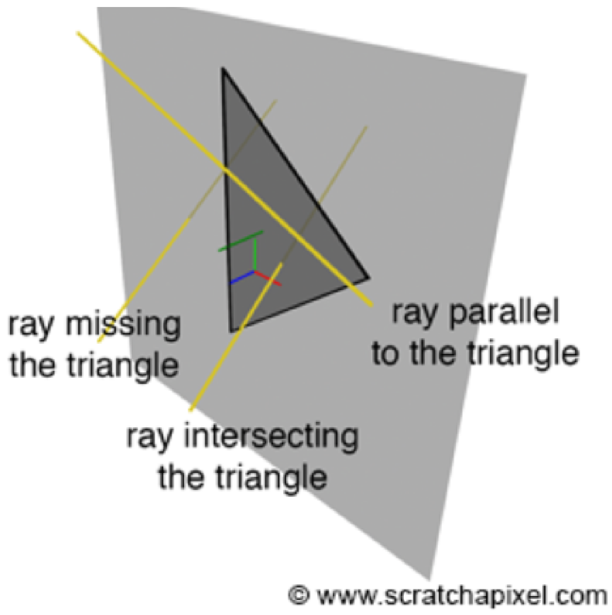
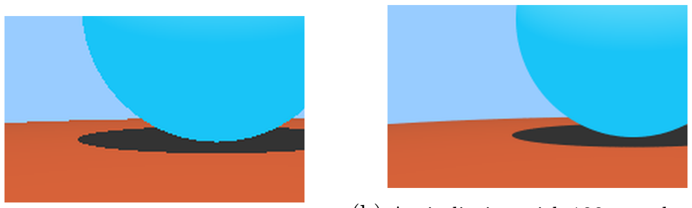


Figure 3: Three cases of ray intersect with a triangle.

2.5 Anti-aliasing

Pixels make up the display on a computer. The smallest component of any digital image is the pixel, and while modern computer monitors have great resolutions with millions of pixels, these pixels are still rectangular. This means that when spherical forms are displayed on a screen, the user will almost certainly notice jagged edges, also known as aliasing. We'll utilize a basic MSAA, which stands for "multisample anti-aliasing" and is one of the most prevalent anti-aliasing techniques. It achieves the finest blend of visual accuracy and performance in most cases. This is done by averaging the number of samples within each pixel. We have several random samples around each pixel, we send rays through each of the samples. The colors of these rays are then averaged.



(a) Alising

(b) Anti-aliasing with 100 samples per pixel

Figure 4: The benefit of using Anti-aliasing improvement, where (b) edges look smoother than (a)

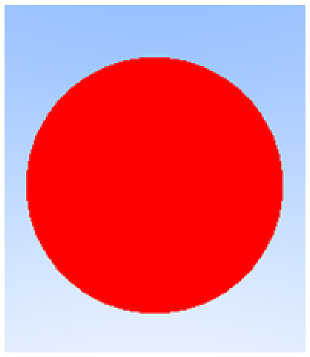
Figure 4 shows the refinement and how the object's edges are smoothed by using the anti-aliasing technique. The issue is performance because the smoother edges get, the more samples had to be taken around each pixel; for example, if the image size is 200 x 200 pixels, and the samples are used for anti-aliasing are equal to 100 sample/pixel, then we will measure the color for 200 x 200 x 100 pixels rather than 200 x 200. A trade-off has to be done here in order to have a smoothed edges, but with no considerable performance cost; this depends on the application and how complex the scene is.

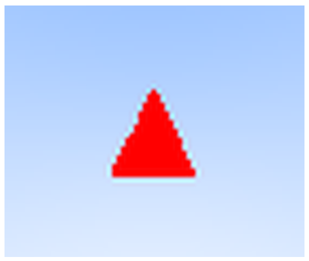
2.6 Implementation

Two classes of Sphere and Triangle are created; each class or model contains parameters depending on its shape, for example the Sphere needs two parameters to represent it: *Radius* and *Center*. Also, each model needs a different intersection test, as explained before, this method will check if the ray intersect the object. After casting rays toward the scene, it will loop over all the objects in the scene and test the intersection; if the ray intersects an object, it will save its position and color of the pixel if there is no object intersect closer to the previous object, we just return the color of the last object pixel. For more complicated objects as a cube, an array or mesh of triangles is arranged in a specific position to compose a cube. This mesh is an object called a cube, but it is composed of triangles; hence the triangle intersection test is being used. For anti-aliasing, there is a configuration class that contains the global settings for the Raytracer, and one configuration is the number of samples used for the anti-aliasing.

2.7 Results and discussion

There are so many formats for images, we will be using PPM file to save the output of the scene. Figure 5 shows different shapes that rendered. Spheres are good models for testing the lights and shading performance, Triangles are helpful in order to create meshes out of them as shown in (c) where a cube is rendered by adding 12 triangles.


(a) Rendering a simple Sphere


(b) Rendering a simple Triangle

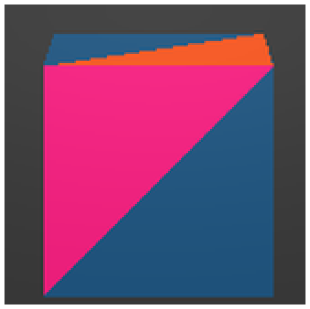

(c) Rendering a cube made out of triangles mesh

Figure 5: Rendering different shapes in the scene

Resources and further reading: Most material and code used in this chapter is mixed between the book [Peter Shirley. *"Ray Tracing in One Weekend"*] and the slides [Matthias Teschner, *"Advanced Computer Graphics"*].

3 Shading

3.1 Introduction

The second step in rendering a scene is *Shading*, and this deals with the color of the object and its intensity. Shading also includes how object's color affects each other; for example, having light hits, the object will make its color look brighter; on the other hand, regions in which light does not hit or reach will have dark color or shadow. In this chapter, shading concepts will be discussed and implemented, in addition to different materials that have different properties and how they interact with the light. The primary key to Shading is calculating the amount of light that hits a point; let us call it P.
The computed light at a point P depends on the following:

- Light illuminated by source $\boldsymbol{L}^{source}$ in real life usually lamp, fire or the sun, it can have any color and intensity but here we will use white color.
- Surface illumination $\boldsymbol{L}^{surface}$.
- Light reflected from the surface $\boldsymbol{L}^{reflected}$.
- The observation angle / looking at angle / camera.

3.1.1 Lambert's Cosine Law

The amount of light energy arriving at a surface is proportional to the cosine of the angle between the light direction and the surface normal, according to *Lambert's cosine law.* Illumination strength at a surface is proportional to the cosine of the angle between $\boldsymbol{l}$ and $\boldsymbol{n}$, the angel will be denoted as θ, the following three cases illustrate the relationship between the $\boldsymbol{L}^{source}$ and $\boldsymbol{L}^{surface}$:
The $\boldsymbol{L}^{surface}$,$\boldsymbol{L}^{source}$ relation is:

$$\boldsymbol{L}^{surface} = \boldsymbol{L}^{source}.\cos\theta \tag{5}$$

- $\boldsymbol{L}^{surface} = \boldsymbol{L}^{source}$, if $\theta = 0°$.
- $\boldsymbol{L}^{surface} = 0$, if $\theta = 90°$.
- $0 < \boldsymbol{L}^{surface} < \boldsymbol{L}^{source}$, if $0° < \theta < 90°$.

3.1.2 Phong reflection model

Phong reflection is a model of local illumination. It defines how light reflects off a surface as a mixture of *diffuse* reflection from rough surfaces and *specular* reflection from polished surfaces. It's based on Phong's intuitive observation that bright surfaces have small, strong specular highlights, and dull surfaces have larger, more gradual specular highlights. The model also includes an ambient term to account for the small amount of light that is scattered about the entire scene.

- **Ambient reflection**

$$\boldsymbol{L}^{amb} = \boldsymbol{\rho} \otimes \boldsymbol{L}^{indirect} \tag{6}$$

  - $\boldsymbol{\rho}$, is the surface color
  - $\boldsymbol{L}^{indirect}$, is the light reflected from other surfaces and objects, excluded the direct light ($\boldsymbol{L}^{source}$)

- **Diffuse reflection**

$$\boldsymbol{L}^{diff} = \boldsymbol{L}^{source}.(\boldsymbol{n}.\boldsymbol{l}) \otimes \boldsymbol{\rho} \tag{7}$$

  - $\boldsymbol{L}^{source}$, is the light source color and intensity which usually white.

– $\boldsymbol{n}$ and $\boldsymbol{l}$, are the representation of the Lambert's cosine law, where $\boldsymbol{n}$ is the normal surface vector and $\boldsymbol{l}$ is the indecent light coming from the light source.

- **Specular reflection**

$$\boldsymbol{L}^{spec} = \boldsymbol{L}^{source}.(\boldsymbol{n}.\boldsymbol{l}).(\boldsymbol{r}.\boldsymbol{v})^m \otimes \boldsymbol{\rho}^{white} \tag{8}$$

– $\boldsymbol{r}$, which is the direction that a perfectly reflected ray of light would take from this point on the surface.

– $\boldsymbol{v}$, which is the direction pointing towards the viewer (such as a virtual camera).

– m, which is a shininess constant for this material, which is larger for surfaces that are smoother and more mirror-like. When this constant is large the specular highlight is small.

The overall illumination on the surface can be computed by summing up the three components that make up *Phong model*:

$$\boldsymbol{L}^{surface} = \boldsymbol{L}^{amb} + \sum_{n=1}^{lights} (\boldsymbol{L}_n^{diff} + \boldsymbol{L}_n^{spec}) \tag{9}$$

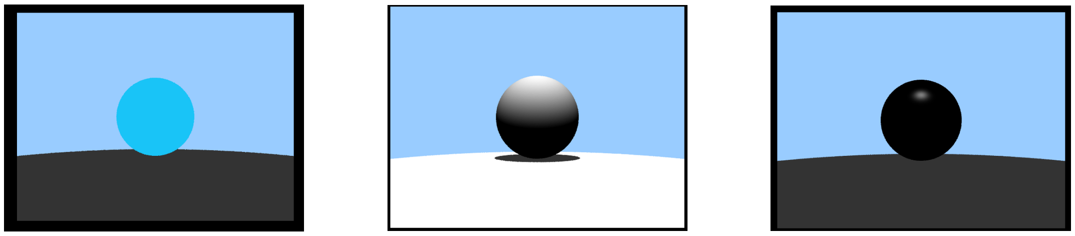
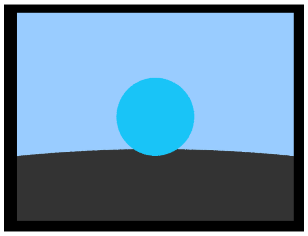
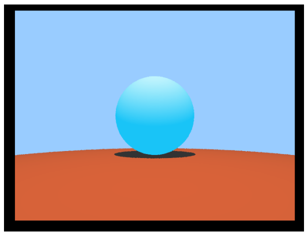

(a) Ambient $\boldsymbol{L}^{amb}$ (b) Diffuse $\boldsymbol{L}^{diff}$ (c) Specular (Glossy) $\boldsymbol{L}^{spec}$

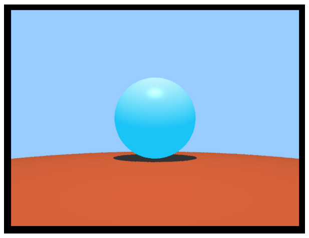
Figure 6: Visual illustration of the Phong equation

3.1.3 Results and discussion

Figure 7 shows the result of using a Phong model to shade a Sphere in a scene. (a) shows the $\boldsymbol{L}^{amb}$ only. (b) represents the $\boldsymbol{L}^{amb}$ and $\boldsymbol{L}^{diff}$, (c) shows the overall light calculated from the scene $\boldsymbol{L}^{amb}$, $\boldsymbol{L}^{diff}$ and $\boldsymbol{L}^{spec}$, (d) illustrated the summation of two different light sources for the $\boldsymbol{L}^{diff}$ and $\boldsymbol{L}^{spec}$ that is why the scene is brighter, two shadows for the Sphere and also two specular points on the Sphere.


(a) Ambient


(b) Diffuse + Ambient


(c) Diffuse + Ambient + Specular (Glossy)

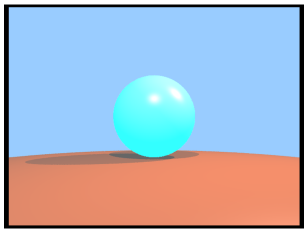

(d) Multiple lights that generates brighter surface and multiple shadows

Figure 7: Using a Phong model to shade a sphere.

3.2 Materials

In Raytracing, one of the important topics is to give the object a material type, and this can be: *Glossy*, *Diffuse*, *Transparent*, and *Subsurface scattering*, each material has different surface reflection and refraction properties, some surfaces reflect the light equally such as diffuse surfaces, some are reflecting light into a dominant direction as Glossy surfaces, some material such as water will reflect some light but also refract some.
In the previous chapter, we discussed diffuse and specular (Glossy) surfaces. In this chapter, we will implement more materials like refraction surface because it has interesting properties that will be illustrated next.

3.2.1 Refraction

The refraction phenomenon happens when the light passes from one medium to a different medium. Figure 8 illustrates this phenomenon, I is the incident light ray, R is the reflected light where N is the normal vector to the surface which is water (in blue), the reflected angle $\theta 2$ is equal to the incident angle $\theta 1$, in addition to the reflected light there is a refracted light T, the direction of T depends on the $\theta 1$ and *refractive index,* η (Describes how fast light travels through the material).
The ratio of the sines of the angle of incidence $\theta 1$ and angle of refraction $\theta 2$ is equivalent to the opposite ratio of the indices of refraction for a particular pair of media, according to *Snell's law*:

$$\frac{sin(\theta_1)}{sin(\theta_2)} = \frac{\eta_2}{\eta_1} \tag{10}$$

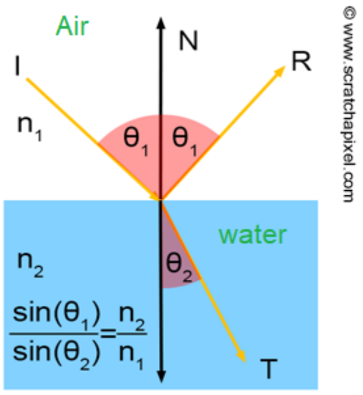


Figure 8: Incident light I in the air hitting water surface, R is reflected light and T is the transmitted and refracted light, n is *refractive index,* η Resource: scratchapixel.com.]

3.2.2 Fresnel

The amount of light that is reflected vs. refracted can be calculated using what is known as the *Fresnel equations*, where F_R is the reflected light portion, and F_T is the portion transmitted through the material.

$$F_R = \frac{1}{2}(\left(\frac{\eta_2 \cos\theta_1 - \eta_1 \cos\theta_2}{\eta_2 \cos\theta_1 + \eta_1 \cos\theta_2}\right)^2 + \left(\frac{\eta_1 \cos\theta_2 - \eta_2 \cos\theta_1}{\eta_1 \cos\theta_2 + \eta_2 \cos\theta_1}\right)^2) \tag{11}$$

$$F_T = 1 - F_R \tag{12}$$

3.2.3 Texture

A texture can be uniforms, such as a brick wall, or irregulars, such as wood grain or marble. The conventional way is to build a "texture map," which is a 2D bitmapped picture of the texture that is then "wrapped around" the 3D object. Instead of using bitmaps, another option is to compute the texture entirely using mathematical models. Textures are so helpful; they reduce the geometric complexity of a scene by mapping a bit directly to an image or having a mathematical equation that can easily represent the surface color, they also reduce the number of vertices, and it reduces the modeling and rendering time.
Procedural texturing describes ways to employ texture values, such as replacing the original surface color with the texture color linearly combining the original surface color with the texture multiply, add, subtract surface color and texture color.

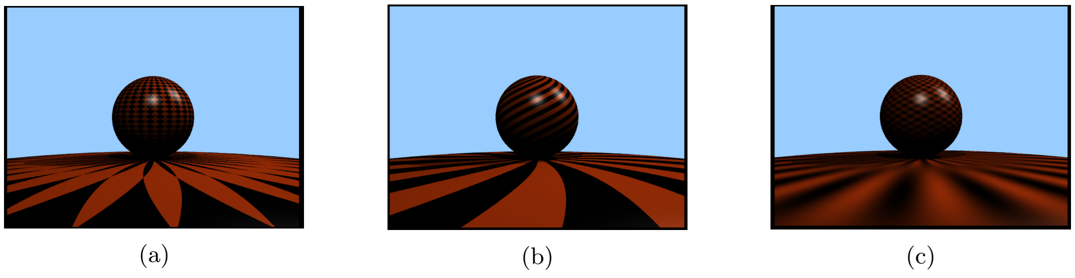


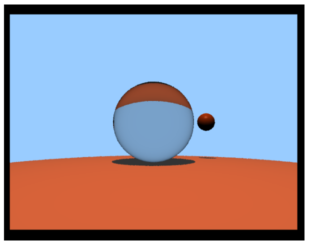
Figure 9: (a), (b) and (c) show different pattern for textures that can be generated by mathematical equations to map x and y value to different color value.

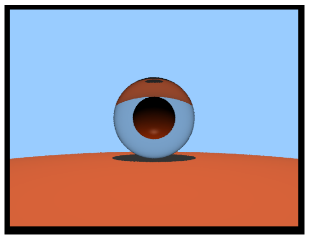
3.2.4 Implementation

Because the Material subchapter is associated with the previous subchapter Shading, both implementations are explained here.
Each object "Sphere" or "Triangle" has a different material type: *DIFFUSE AND GLOSSY*, *REFLECTION AND REFRACTION* , *TEXTURE* . In the shading process, the type is being checked, and depending on it; a switch case is used to shade the object accordingly.
For *DIFFUSE AND GLOSSY* material, a Phong model equation is used to calculate the pixel's color; *REFLECTION AND REFRACTION* is using Fresnel law, *TEXTURE* is using three different patterns chosen randomly.

3.2.5 Results and discussion

Figure 10 shows the refraction phenomenon where the settings had a red glossy sphere passing behind a transparent sphere with an refractive index, η not equal to the air we are assuming it is any kind of liqued with big η that it refract the light with a big angle. As it can be noticed the red sphere looks larger than it should be and also inverted, moreover the ground is inverted as well,


(a) Glass sphere refracts the inverted ground upside down, the red sphere looks normal before refraction


(b) The glass sphere refracts the ground, and the red Sphere directly behind it is inverted

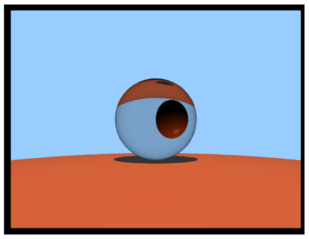

(c) Glass sphere refract the ground, and red Sphere behind it inverted with an angle

Figure 10: The scene shows a transparent sphere that refract the light, where it refract the light and shows the ground and the red sphere inverted.

Resources and further reading: Most material and code used in this chapter is mixed between the website [Jean-Colas Prunier , *"Scratchapixel.com"*] and the slides [Matthias Teschner, *"Advanced Computer Graphics"*].

4 Curves

4.1 Introduction

We frequently need to draw different shapes and models of things onto the screen in computer graphics. Models are not always flat, and we must draw curves many times to draw a smooth model; for example, drawing the famous Utah teapot, this complex model is difficult to draw by only using triangles as the surface of the model needs a curved line than flat, straight lines.

Curves have three categories: *Explicit*, *Implicit*, and *Parametric curves.* In practice, parametric curves are used; hence in this chapter, parametric curves are used.
It only requires 4 points to create a Bézier curve. These points are control points defined in 3D space. As with surfaces, the curve itself does not exist until these 4 points are combined and weighted with some coefficients. Curves defined by parametric equations have a parameter, which is a variable in this equation used to define the curve. To achieve a smoother result, increase the number of segments and points. Font modeling, animation, and games use curves to smooth surfaces and make the scene looks natural rather than looking edgy.

4.2 Bézier Curves

They are a simple and intuitive representation of curves. They are polynomial curves represented by control points where $n + 1$ control points are needed for a curve of degree n. Interpolation is used for the first and last control points; other control points are approximated. The next equation represents the control points depending on the degree n:

$$\boldsymbol{x}(t) = \sum_{i=0}^{n} B_{i,n}(t)\boldsymbol{P}_i,\ t \in [0, 1] \tag{13}$$

Where $\boldsymbol{P}_i$ is the set of points and $B_{i,n}(t)$ represents the Bernstein polynomials that shown in equation 14, Where n is the polynomial degree, i is the index, and t is the variable.

$$B_{i,n}(t) = \frac{n!}{(n-i)!i!}(1-t)^{n-i}t^i,\ \ 0 \le i \le n \tag{14}$$

4.3 Implementation

In this scenario, the curve is represented by a thin cylinder or a long string (like spaghetti), let us take a challenging curved object to render like hair.
Hair rendering steps:

- Create Position at Regular Intervals: It is first necessary to create a loop of vertices along the curve and then connect these vertices together to form faces.
- Create a Local Coordinate System to Generate a Loop of Vertices: we will create a local coordinate system which we will need in step 3.
- Generate loop of vertices: It is making loops with points that are oriented correctly around curves.
- Meshing: By connecting the vertex points, we can form the faces.

4.4 Results and discussion

Figure 11 shows the final result after following the four points mentioned in the implementation, where rendering a hair as a showcase to implement curves is successful. Using more surfaces to render a cylinder makes it smoother as shown in case (a) where 16 surfaces have been used, (b) is using fewer surfaces as it uses 8, however (c) is using only 4, that is why it looks edgy. Increasing the number of surfaces depending on the quality of application; for example, for more extensive scenes, maybe choosing fewer surfaces is better to increase the performance, but for a high-quality rendering, the more surfaces, the better.

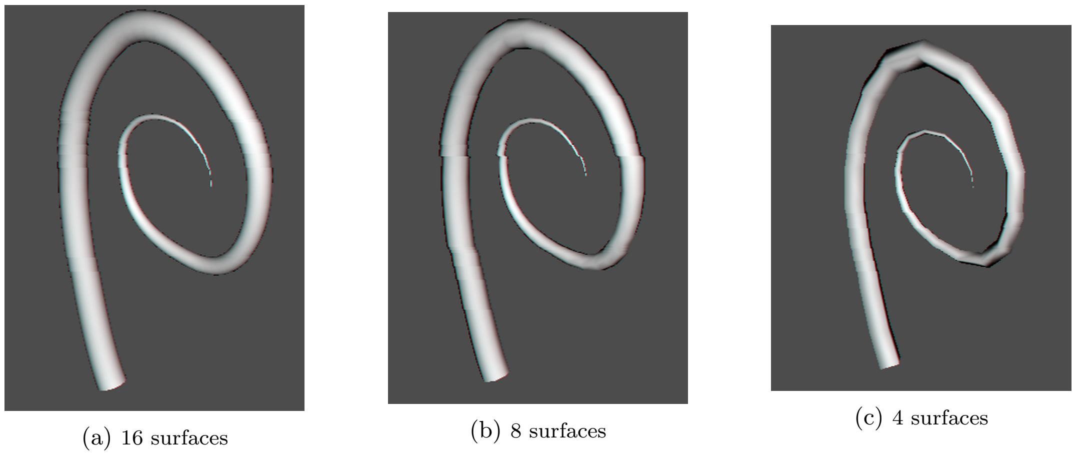

(a) 16 surfaces (b) 8 surfaces (c) 4 surfaces

Figure 11: Hair rendering result for different surfaces numbers of the cylinders

As mentioned before, the higher the order of the polynomial used in the Bézier Curve, the smoother the curve can be. Figure 12 shows the differences between using a polynomial of order three and two. As it can be noticed, (a) has smoother curves than (b) because (a) is using a polynomial of order three, unlike (b) that uses only two, and the higher the order of the polynomial the more complex curves can be represented, however the more Bernstein parameters to be calculated and this will reduce the performance of the Raytracer.

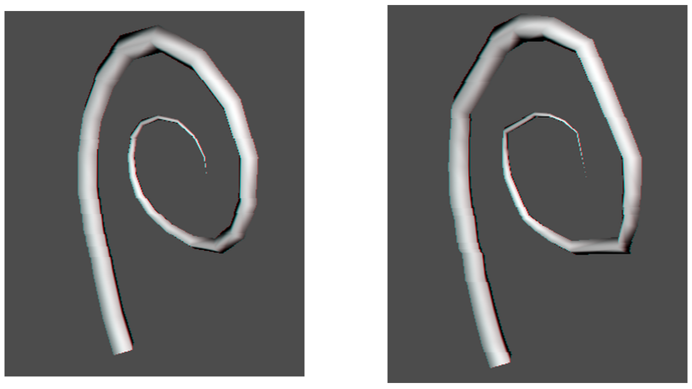

(a) 3^{rd} order (b) 2^{nd} order

Figure 12: Hair rendering result for different polynomial order.

Resources and further reading: Most material and code used in this chapter is mixed between the book [Peter Shirley. *"Ray Tracing in One Weekend"*] and the slides [Matthias Teschner, *"Advanced Computer Graphics"*].

5 Bounding Volume Hierarchies - BVH

5.1 Introduction

Bounding Volume Hierarchies, known as "BVH," is simply a data structure representing complex geometric models with specific simple bounding volumes to reduce some of the expensive tests in different computer graphics applications. BVH is object-oriented, unlike other algorithms such as kd-trees which are space subdivisions.

5.2 Motivation

There are two main applications for bounding volume hierarchy, in retracing where two main challenges have to be solved: Shading and visibility; usually the Raytracer has to test each model in the scene in order to render it to the screen, on the other hand, some of the models they are not visible to the camera and testing them make no sense therefore by telling the rays which models to test the intersection this will boost the performance of the Raytracer tremendously, BVH can quickly achieve this.
Moreover, the collision detection algorithm used in simulation and games can benefit significantly from BVH.

5.3 Bounding volume - BV

Bounding volume is the tightest possible virtual volume that wraps up a model in a scene. It is the fundamental component that builds the BVH tree.
There are four different main types of BV depending on the shape complexity:

- Spheres.
- Axis Aligned Bounding Box (AABB).
- Oriented Bounding Box (OBB).
- Discrete Oriented Polytope (k-DOP).

5.4 BVH Tree construction

Nodes in the BVH tree are BV, and leaves are primitives. There are three different ways to construct a BVH tree:

- *Top Down*: Arguably the most popular technique in practice. It uses the '*fit and split*' algorithm, where it starts with the whole model and encapsulates it with a BV and fits it, then tries to split it into n children, usually two. It keeps recursively splitting and fitting until it reaches the leaves and assigns the primitives to them.
- *Bottom Up*: Slower construction time than *Top-Down* but usually produces the best tree. It uses the '*knit and fit*' algorithm, where it starts with primitives and tries to encapsulate them with the BV and merge each n, usually $n = 2$; it keeps recursively doing this process until it reaches the root.
- *Insertion*: It uses '*incremental-insertion*' algorithm, where it starts with a single leaf and merges another leaf by using a cost function, then creating a head node and searching for the next leaf to merge. The problem with this method is that it can become worst as it depends on the insertion order of the nodes, and it is challenging to find the best tree.

5.5 Implementation

In this Raytracer, a Sphere BV is used; the reason is Sphere has a fast intersection test and is easy to generate and fit into the model or primitives. On the other hand, Sphere is usually not so tight, resulting in non-optimal performance, but for this Raytracer, as the scene is not significant and complex, Spheres can be a good choice.

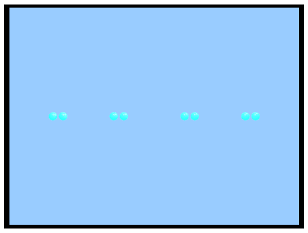
For constructing the BVH tree, the Bottom-Up method is used. Moreover, a binary tree and minimum leaves = 2 are the settings. Eight particles in the scene are generated for testing, these particles are the leaves, each particle is a leave node that don't have children, each two particles are merged into a Sphere BV parent node that has a centre of a middle point $c_{1,2} = \frac{c_1+c_2}{2}$ and radius of both particle radius combined $r_{1,2} = r_1 + r_2$. We loop over this process until no object left and we reach the root of the scene, and we return the root reference. For ray intersection test, the next algorithm explain the process:

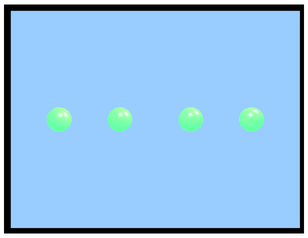
Algorithm 1 Raytracer intersection test by using BVH algorithm

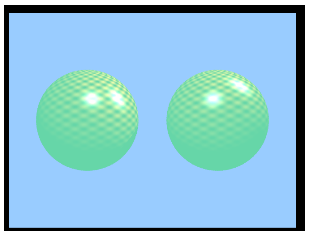
```
 1: procedure MyProcedure
 2:     Beginning at the root node BV
 3:     if Ray dose not intersect the root BV then return No intersection
 4:     else
 5:         check all children of the node
 6:         if there is intersection between any child with Ray then
 7:             if leaf node then return Intersection might happened
 8:         else
 9:             goto 4.
10:     if No children left then return No intersection
```

5.6 Results and discussion

Figure 13 (a) shows eight particles that will be used as a test scene; these particles are the leaves in our tree that present layer 4. Because we are using a bottom-up strategy, every two particles are merged and encapsulated by one Sphere BV. This creates a higher level of the tree, layer 3. Recursivity we keep merging every two BVs until we hit the root, as shown in (d), note that in practice we do not render the BV spheres they are virtual bounding volume, but I am rendering them for illustration only..


(a) Level 4: Particles that will be encapsulate by BV


(b) Level 3: Sphere BV wraps up particles


(c) Level 2: Second level of the BVH tree

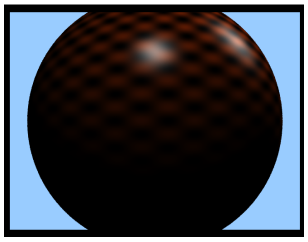

(d) Level 1: This is the root node the cover all the BVs and particles

Figure 13: BVH result when a Sphere BV used and the scene is eight particles

In order to test the BVH performance, three main tests have been done. Table 1 illustrates the differences between using BVH or not. By looking into the table, we noticed that using the BVH spends double the time of not using BVH. The main reason is that BVH adds more steps to the Raytracer; it adds tree Construction cost and adds BV generation cost. BVH is useful when more extensive scenes are used; for example, by including 16 particles in the scene and excluding the other 16 particles from the camera vision, we can see that BVH boosts the Raytracer's performance, because half of the objects are outside the camera vision therefore less intersection tests are made.

-	8/8	16/16	16/32
With BVH	7s	10s	10s
Without BVH	3s	7s	12s

Table 1: The Raytracer performance comparison, where 16/32 means 32 particles are used in the scene and only 16 are visible to the camera.

6 Summary and conclusion

In the first chapter, *Visibility*, we introduced different shapes that can be represented in the 3D scene; it can be noticed that some are easier to be rendered than others depending on how complex the model is, but also the intersection test plays a considerable role for the performance. Also, some complex shapes like tetrahedra and cubes can be reconstructed by building up a trivial model as triangles.
Moreover, rendering pixels can lead to aliasing where the edges of the model look sharp and not realistic; this can be solved by using different anti-aliasing methods; however, this can reduce the performance of the Raytracer.

The *Shading* chapter Phong model is used to approximate the light effect on Spheres where diffuse and glossy models are used; also, different materials such as transparent have been implemented and shown using snells law. Textures are a simple method to shade a model without using expensive operations, and a math model can give excellent results.

In Shading, the more Rays paths are traced for rendering, the more realistic the scene becomes; however, the deeper we trace the Rays, the more expensive the Raytracer becomes as it needs more paths to trace, this can be improved by *Monte carlo approximation*, however this is out of this lap scope.

Curves are not trivial to be approximated in computer graphics because we approximate curves by lines; hence, the smoother the curve, the more lines are needed. As we introduced the polynomials, if we want a complex curve, we need a higher-order polynomial, which leads to higher memory consumption and reduces the Raytracer's performance.

As raytracing operations are expensive and more paths to be traced, some data structures are used to increase the Raytracer performance and efficiency, some methods are space subdivisions as kd-trees, and others objects oriented as BVH. BVH can reduce the number of intersection tests needed; however, it adds more steps to the workflow as tree construction, tree updates, and tree traversal. BVH can be optimized by tweaking the BV types used, tree construction algorithms, and traversal algorithms.

In this report, different resources are used for the implementation to search and test different approaches of Raytracer. The report illustrates the basic approaches and challenges that can encounter the engineer to solve the rendering challenges.

References

[1] Fangkai Y. *"Collision Detection in Computer Games"*. KTH Royal Institute of Technology

[2] Hamzah S, Abdullah B. *"Bounding Volume Hierarchies for Collision Detection"*

[3] Jean-Colas Prunier, *"Scratchapixel.com"*

[4] Matthias Teschner, *"Advanced Computer Graphics"*, University of Freiburg

[5] Matthias Teschner, *"Simulation in Computer Graphics Bounding Volume Hierarchies"*, University of Freiburg

[6] Peter Shirley. *"Ray Tracing in One Weekend"*

[7] Peter Shirley. *"Ray Tracing: The Next Week"*

[8] Piotr Dubla. *"Interactive Global Illumination on the CPU"*

[9] Stefan G. *"Collision queries using bounding box"*. The University of North Carolina