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

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

July 11, 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. In this report, we will be using Raytracing. 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.

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. It naturally occurs in rendering because depicting invisible objects is both inefficient and wrong. This challenge is referred to as visible surface determination or concealed surface removal, depending on how it is approached. The primary two operations in the Raytracer are Rays casting and Path tracing.

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. All raytracers use rays as a way to simuate photons. Let's think of a ray as a function

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

Here is a 3D position along a line in 3D. $\boldsymbol{o}$ is the ray origin and $\boldsymbol{d}$ is the ray direction.

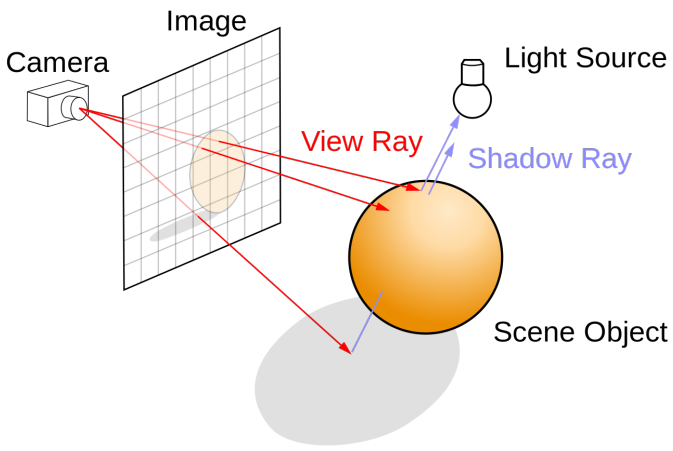


Figure 1: The raytracing algorithm builds an image by extending rays into a scene and bouncing them off surfaces and towards sources of light to approximate the color value of pixels [Piotr Dubla, "Interactive Global Illumination on the CPU."]

2.3 Rendering a 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:

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

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

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

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

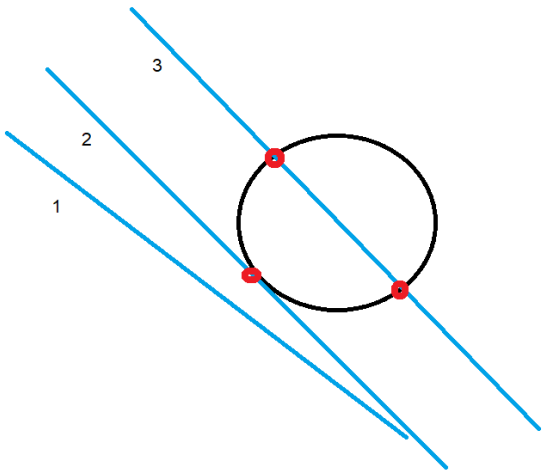


Figure 2: The three possible line-sphere intersections: 1. No intersection. 2. Point intersection. 3. Two point intersection.

2.4 Rendering Triangle

It is computing the intersection of a ray with a primitive such as a sphere is not tricky. However, since it is challenging to model most 3D objects with spheres alone, it is necessary to use some other types of primitive to represent more complex objects (objects of arbitrary shape). Instead of working with complex primitives such as NURBS or Bezier patches, we can convert every object into a triangle mesh and compute the intersection of a ray with every triangle in this mesh. We get three different cases by solving the equation as shown in Figure 3 below.

Parametric representation:

$$\boldsymbol{p}(b_1, b_2) = (1 - b_1 - b_2)\boldsymbol{p}_0 + b_1\boldsymbol{p}_1 + b_2\boldsymbol{p}_2 \tag{3}$$

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 the one we are interested in is when the ray intersects, we get an intersection if: $b_0 \geq 0 \wedge b_1 \geq 0 \wedge b_2 \geq 0$

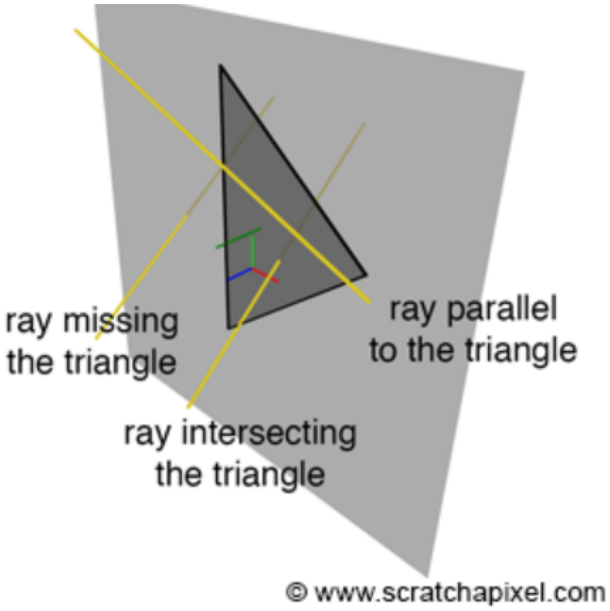


Figure 3: Three cases of ray intersect with a triangle.

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 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 a number of samples within each pixel. We have several samples within that pixel for a given pixel and send rays through each of the samples. The colors of these rays are then averaged.

(a) (b)

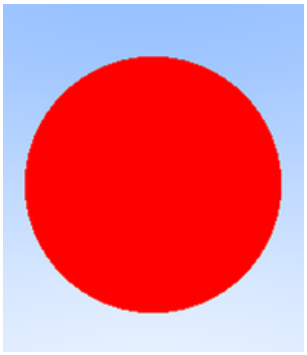
Figure 4: (a)Alising, (b) Anti-aliasing with 100 samples per pixel.

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 more smoothed edges get, the more samples had to be taken around

each pixel; for example, if the image size is 200 x 200, and the samples are used for anti-aliasing are equal to 100, 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.

2.6 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

3 Shading

3.1 Introduction

Rendering a scene needs two steps; the first step is solving the visibility issue, which means which object is visible to the camera and what its shape. The second step 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. 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 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: $\boldsymbol{L}^{surface} = \boldsymbol{L}^{source}.\cos\theta$

- $\boldsymbol{L}^{surface} = \boldsymbol{L}^{source}$, 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 intuative observation that bright surfaces have small, strong specular highlights, and dull surfaces have larger, more gradual specular highlights. nThe model also includes an ambient term to account for the small amount of light that is scattered about the entire scene.

- **Ambient reflection**

$$\boldsymbol{L}^{amb} = \boldsymbol{\rho} \otimes \boldsymbol{L}^{indirect} \tag{4}$$

  - $\boldsymbol{\rho}$, 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**

$$\boldsymbol{L}^{diff} = \boldsymbol{L}^{source}.(\boldsymbol{n}.\boldsymbol{l}) \otimes \boldsymbol{\rho} \tag{5}$$

  - $\boldsymbol{L}^{source}$, 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 n is the normal surface vector and l is the indecent light coming from the light source.

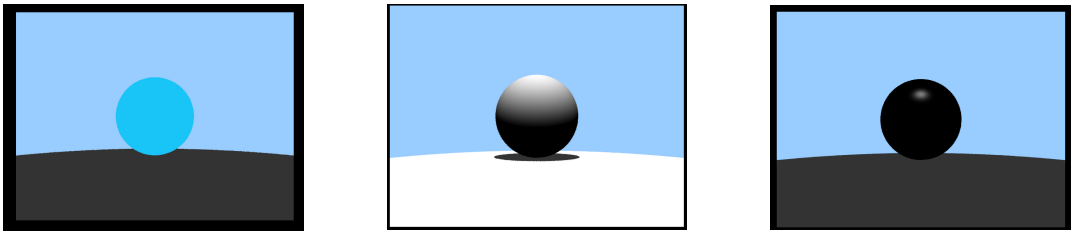
- **Specular reflection**

$$\boldsymbol{L}^{spec} = \boldsymbol{L}^{source}.(\boldsymbol{n}.\boldsymbol{l}).(\boldsymbol{r}.\boldsymbol{v})^{m} \otimes \boldsymbol{\rho}^{white} \tag{6}$$

  - $\boldsymbol{r}$, 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*:

$$\boldsymbol{L}^{surface} = \boldsymbol{L}^{amb} + \sum_{n=1}^{lights} (\boldsymbol{L}_n^{diff} + \boldsymbol{L}_n^{spec}) \tag{7}$$

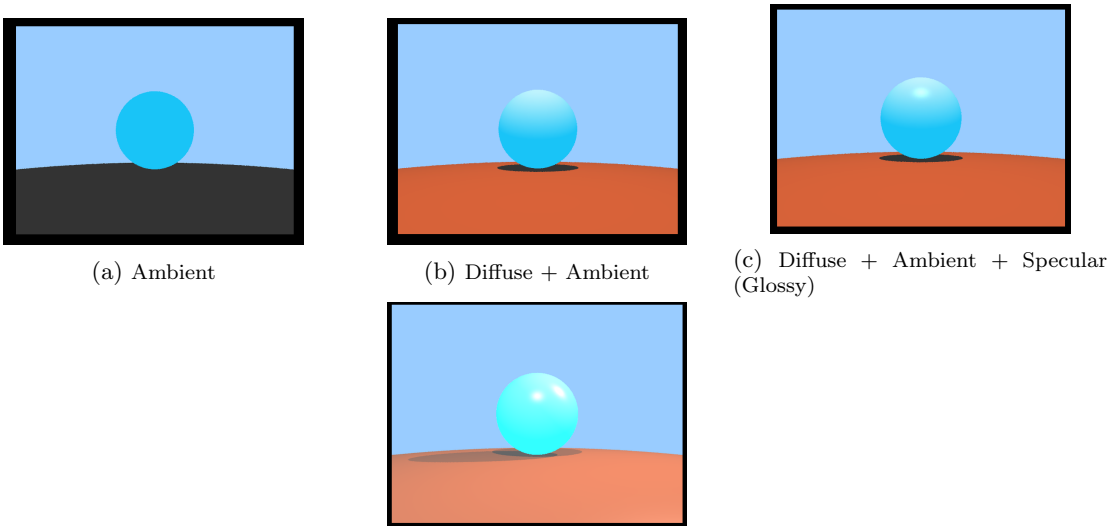


(a) Ambient (b) Diffuse (c) Specular (Glossy)

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 a refraction surface because it has interesting properties.

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*:

$$\frac{sin(\theta_1)}{sin(\theta_2)} = \frac{\eta_2}{\eta_1} \tag{8}$$

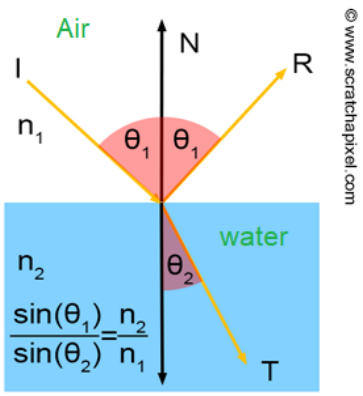


Figure 8: Incident light I in the air hitting water surface, R is reflected light and T is the transmitted and refracted light, n is *refractive index*, η Resource: scratchapixel.com.]

3.2.2 Fresnel

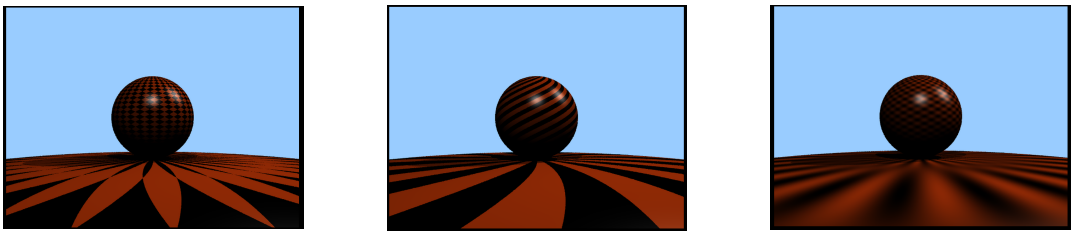
The amount of reflected vs. refracted light can be computed using what we call the *Fresnel equations*, where F_R is the reflected light portion, and F_T is the portion transmitted through the material.

$$F_R = \frac{1}{2}\left(\left(\frac{\eta_2 \cos\theta_1 - \eta_1 \cos\theta_2}{\eta_2 \cos\theta_1 + \eta_1 \cos\theta_2}\right)^2 + \left(\frac{\eta_1 \cos\theta_2 - \eta_2 \cos\theta_1}{\eta_1 \cos\theta_2 + \eta_2 \cos\theta_1}\right)^2\right) \tag{9}$$

$$F_T = 1 - F_R \tag{10}$$

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.

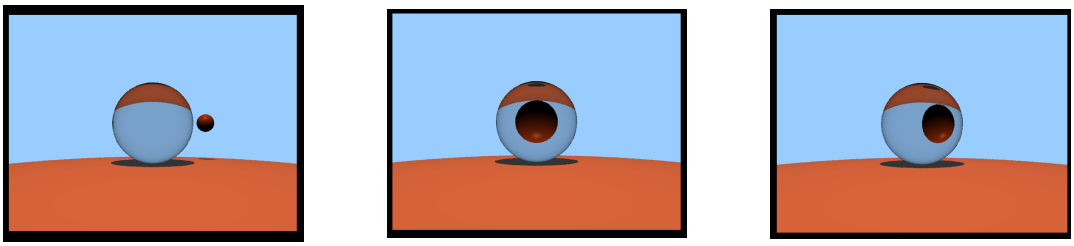


(a) (b) (c)

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 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) (b) (c)

Figure 10: (a). Glass sphere refract the ground inverted upside down. (b). Glass sphere refract the ground and red sphere directly behind it inverted. (c). Glass sphere refract the ground and red sphere behind it inverted with an angle

4 Curves

4.1 Introduction

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:

$$\begin{aligned} \boldsymbol{x}(t) &= \sum_{i=0}^{n} B_{i,n}(t)\boldsymbol{P}_i, \ t \in [0,1] \\ B_{i,n}(t) &= \frac{n!}{(n-i)!i!}(1-t)^{n-i}t^i, \ \ 0 \leq i \leq n \end{aligned} \tag{11}$$

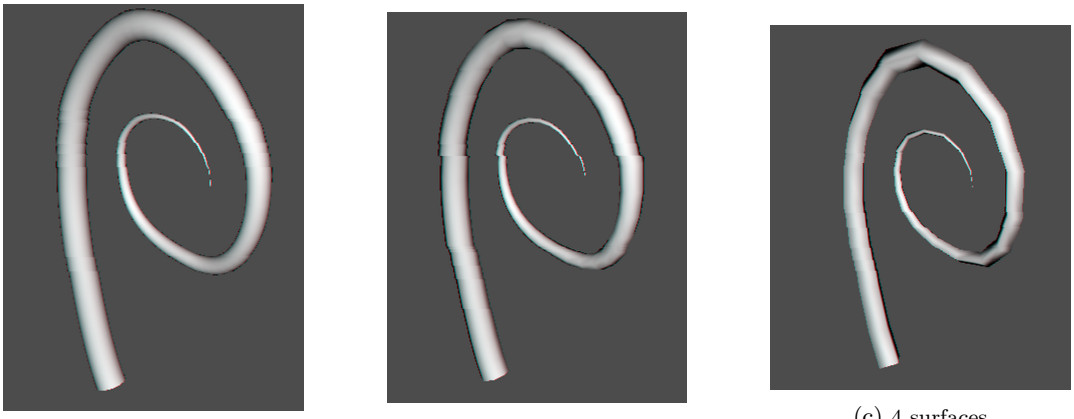
4.3 Implementation

In this scenario, the curve is represented by a thin cylinder or a long string (like spaghetti).
Hair rendering:

- 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

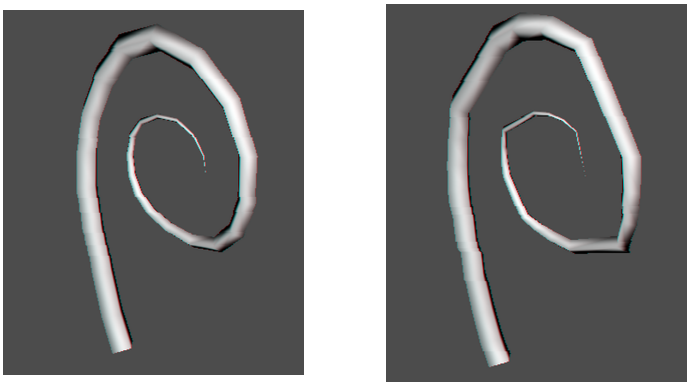
Figure 11 shows the final result after following the four points mention in the implementation, where rendering a hair as a showcase to implement curves is successful. Using more surfaces to render to cylinder makes it smoother as shown in case (a) where 16 surfaces have been used, (b) is using fewer 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.



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

Figure 12: Hair rendering result for different polynomial order.

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:

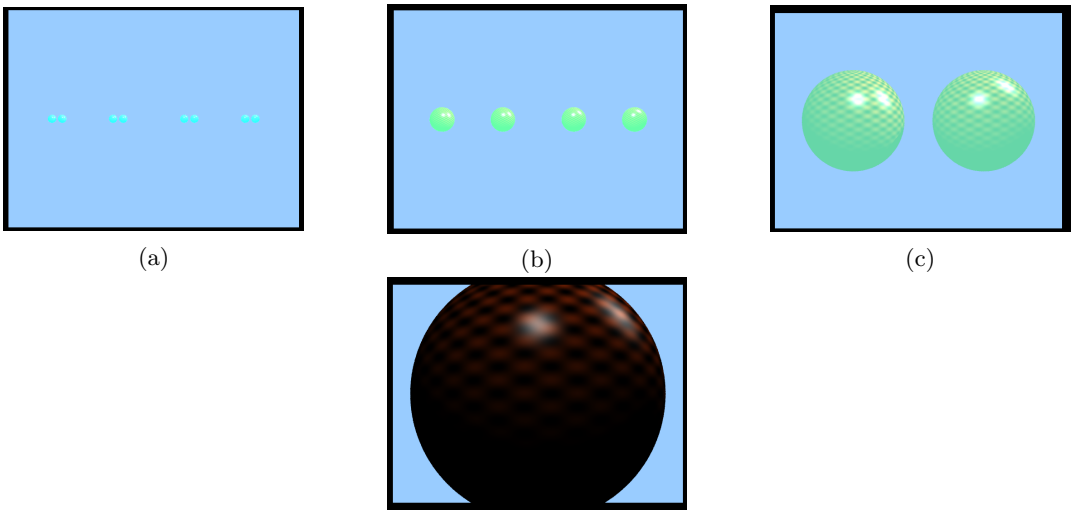
- Top Down: Arguably the most popular technique in practice. It uses the 'fit and split' algorithm, where it starts with the whole model and encapsulates it with a BV and fits it, then tries to split it into n children, usually 2. It keeps recursively splitting and fitting until it reaches the leaves 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 simple as Sphere has fast intersection text 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.

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).



(a) (b) (c)

(d)

Figure 13: (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.

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.

-	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 sceen and only 16 are included in the image.

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

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

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

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

[7] Scratchapixel.com, *"Learn Computer Graphics From Scratch!"*

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