

Final Report

|  |
| --- |
| School of Computing  Faculty of Engineering AND PHYSICAL SCIENCES |

<Effective curve primitive rendering in ray tracer>

<Hanmun Hwang>

< BSc Computer Science >

**<2023/24>**

**<** *COMP3931 Individual Project* **>**

The candidate confirms that the following have been submitted*:*

|  |  |  |
| --- | --- | --- |
| **Items** | **Format** | **Recipient(s) and Date** |
| Final Report | PDF file | Uploaded to Minerva (01/05/24) |
| Link to online code repository | URL | Sent to supervisor and assessor (01/05/24) |

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student) Hanmun Hwang



© <Year of Submission> The University of Leeds and <full name of candidate>

# Abstract

Drawing simple geometries, such as spheres and triangles, is relatively straightforward when the vertices that form them are known. However, realistic camera imaging in 3D graphics involves more complex considerations, such as ‘How are 3D appearances captured on a 2D screen?’ and ‘How is the Interaction between rays and objects represented ?’. Here is a 3D graphic technique called ‘Ray tracing’ designed to address these challenges effectively and simulate interactions with light ray within a window screen. By using the rendering of Bezier surface as foundation for complex shapes, this discussion will provide an insight into how surface based on Bezier curves is integrated into the 3D space of ray tracing.

From the output, it was observed that both curve and surface are approximated or tessellated into simpler geometric primitives, such as triangles, to facilitate the computation of ray-object intersections. Finally, it was explored that for high performance, more rays are casted into the scene although rendering time increased significantly.

# Acknowledgements

I would like to thank my supervisor, Markus Billeter, for his great support and guidance to choose the project. His expertise about what this project aims to helped me a lot.

I would also like to thank my assessor, Vania Dimitrova, for her great feedback that pointed out what my work was lacking. Her advice helped me ensure that the project report covers what they are looking for.

I appreciate to all those who showed me their support and encouraging during challenging times for this project.

# Contents

[Abstract iii](#_Toc165435578)

[Acknowledgements iv](#_Toc165435579)

[Introduction and Background Research 1](#_Toc165435581)

[1.1 Ray Tracing 1](#_Toc165435582)

[1.1.1 Ray 2](#_Toc165435584)

[1.1.2 Rendering pixel by pixel 2](#_Toc165435584)

[1.1.3 Intersection 3](#_Toc165435585)

[1.1.4 Shading 4](#_Toc165435586)

[1.1 Bezier Curve 6](#_Toc165435587)

[1.2.1 Mathematical Concept 6](#_Toc165435588)

[1.2.2 Bezier Surface (Mesh modelling) 9](#_Toc165435589)

[Methods 10](#_Toc165435591)

[2.1 Development tools 10](#_Toc165435592)

[2.2 Ray-tracer construction 10](#_Toc165435593)

[2.2.1 Rendering pipeline 10](#_Toc165435594)

[2.2.2 Rendering an image 11](#_Toc165435595)

[2.3 Bezier processing integration 15](#_Toc165435596)

[2.3.1 Integration ray-tracer with Bezier curve 15](#_Toc165435597)

[2.3.2 Fetch Implementation using curves 17](#_Toc165435598)

[Results and Improvement 21](#_Toc165435599)

[3.1 Test with Bezier curve modelling 21](#_Toc165435600)

[3.2 Improve image performance 21](#_Toc165435601)

[3.2.1 Aliasing 21](#_Toc165435602)

[3.2.2 Shadow acne 23](#_Toc165435603)

[3.3 Test with Bezier Surface modelling 24](#_Toc165435604)

[3.4 Model evaluation 25](#_Toc165435605)

[3.4.1 Rendering time 25](#_Toc165435606)

[3.4.2 Analysis 25](#_Toc165435607)

[3.4.3 Plans for its solution 26](#_Toc165435608)

[Discussion 27](#_Toc165435609)

[4.1 Limitations 27](#_Toc165435610)

[4.2 Conclusions 27](#_Toc165435611)

[4.3 Ideas for future work 28](#_Toc165435612)

[4.3.1 Real-time Ray tracing 28](#_Toc165435613)

[4.3.2 Path tracing 28](#_Toc165435614)

[List of References 29](#_Toc165435615)

[Appendix A Self-appraisal 31](#_Toc165435616)

[A.1 Critical self-evaluation 31](#_Toc165435617)

[A.2 Personal reﬂection and lessons learned 31](#_Toc165435618)

[A.3 Legal, social, ethical and professional issues 32](#_Toc165435619)

[A.3.1 Legal issues 32](#_Toc165435620)

[A.3.2 Social issues 32](#_Toc165435621)

[A.3.3 Ethical issues 32](#_Toc165435622)

[A.3.4 Professional issues 32](#_Toc165435623)

[Appendix B External Materials 33](#_Toc165435624)

[B.1 Code design 33](#_Toc165435625)

[B.2 Tools 33](#_Toc165435626)

# Chapter 1

### Introduction and Background Research

## Ray Tracing

Adjusting a sequence of pixels for rendering common geometries in window is not enough to generate a realistic image. When taking a photograph with a camera, does the image appear as a two-dimensional flat plane? No, while the photograph itself is a two-dimensional representation, modern cameras are designed to capture depth, texture, and perspective, which give the image a sense of three-dimensionality. How about taking a picture at night without a flashlight? It will likely turn out black. Raytracing is a computer graphic technique that emulates the way light reflects and refracts in the real world, providing a more realistic and believable environment.

As we know Intuitively, this technique is literally “Tracing a ray”. What comes in our thought when we see a thing is to stare at something but from the perspective of computer graphics or physics, it is of ‘light detection’. Rather than a specific substance or wave from the eye to detect an object, light scattered in all directions is reflected or refracted and enters our eyes. When we chase the ray reversely, it is possible to explain where the source of ray is and how an object looks like once detected something. In other words, by tracing the light that reaches the human eye back, it becomes to mimic how the object should appear in that situation(see Figure 1).

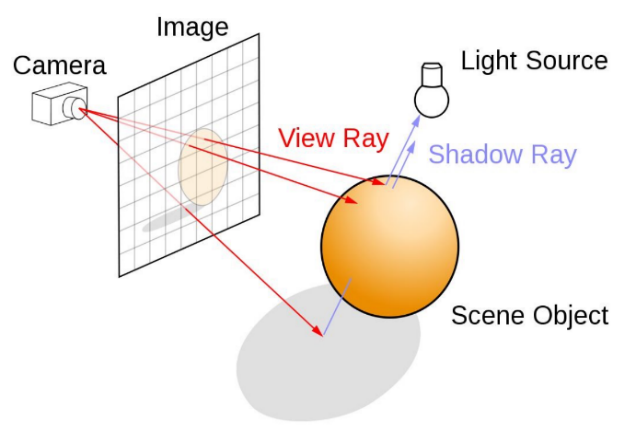


Figure 1

In 1979, Turner Whitted [7] firstly introduced a method for computing ray reflections and refractions using ray tracing techniques, marking a significant advancement in the creation of realistic images in computer graphics. This approach laid the groundwork for simulating complex optical effects such as shadows, reflections, and refractions, thereby enhancing the visual fidelity of computer-generated imagery. His work enables developers to simulate the lighting of a scene and its object with diverse effects.

### Ray

A ray symbolizes essentially a beam of light. Just as our eyes perceive the surrounding world by receiving Rays emitted from various sources, a ray-tracer uses ‘ray’ to simulate the interactions between light and virtual models, presenting them through the viewport or camera. Rays enter the camera from the respective pixel, calculating and simulating the path that light takes. The ray can be mathematically represented as a point defining the ray’s starting position in space, along with a directional vector indicating its direction.

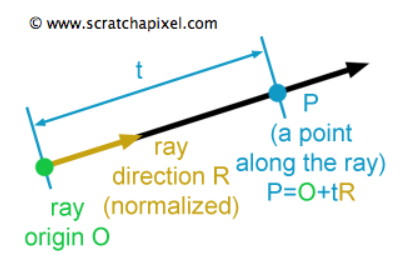


Figure 2

(equation 1.0)

P : A position along the line in 3D

O : Ray origin

: Ray direction

t : Ray parameter that scales

The parameter t in this equation can either be positive or negative. If **t** value is positive, the point **P** in front of the ray’s origin point. The other way if **t** is negative, the point **P** is behind the ray origin. In general ray-tracer, the intersection points between ray and surfaces always be in front of the ray’s origin that means we do not consider situations occurring behind the ray's origin.

### 1.1.2 Rendering pixel by pixel

In a similar manner like the natural propagation of light from sources, ray-tracer casts primary rays(see equation 1.0) from camera origin into the scene, rendering pixel by pixel based on the geometries these rays intersect with. To determine the direction of each ray, a virtual viewport or pixel frame is utilized, facilitating the precise tracking of each ray's path(see Figure 3).

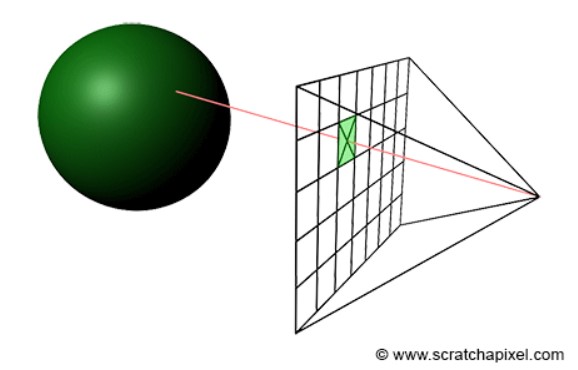


Figure 3

A primary ray passes through each pixel on the grid frame by which it requires to know the coordinate of middle within each pixel. For calculating the middle of a pixel, unit u and v vectors are used in this ray-tracer(see Figure 4).

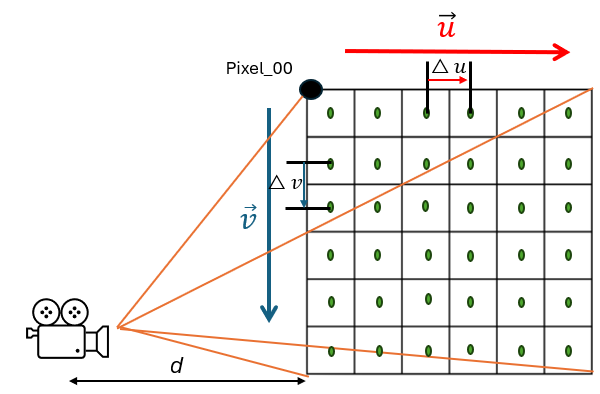
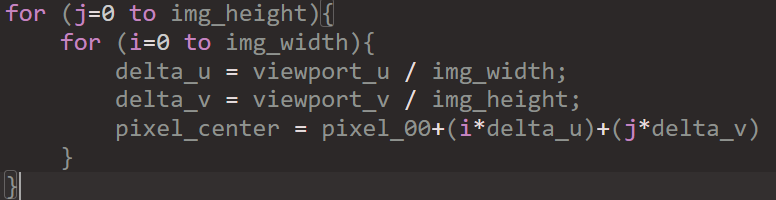


Figure 4



### 1.1.3 Intersection

What is the concept of ray-intersection? it is a process by which rays determine visible surfaces by intersecting with scene geometries. In this part, a sample geometry is needed to explain and calculate whether a ray hits or not. Sphere is a decent option because it is relatively simple which is best. Looking back in mathematic class, we have learnt the equation of a sphere as shown below (see equation 1.1).

(equation 1.1)

As for 3D graphics, those need to be converted into vectors so that x, y, z axis is under the hood in the Vec3 class(Used to represent vectors in 3d space). And we will utilize above equation for determination of ray-object intersection(see Figure 5).

Vector : It is as is the vector .

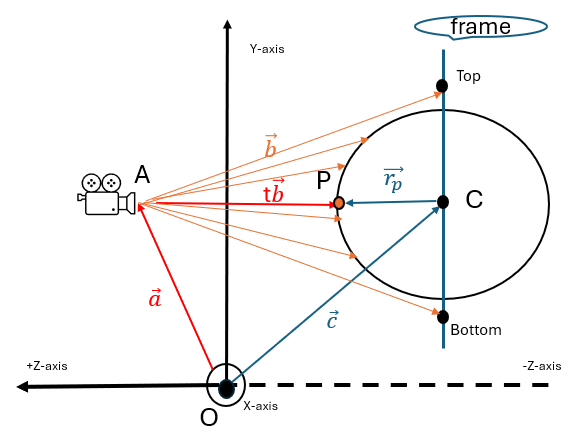


Figure 5

Vector : Vector camera to the viewport.

Vector : Sphere centre to a potential intersection point P.

Vector t : If there were an intersection along the vector would be t (t is real number).

A math equations with numbers and symbols

Description automatically generated

The left solving process for the math equation looks hard to understand, but the vectors and radius in that equation are all constant and known. The only unknown is t, and we have a , which means that this equation is quadratic. It is possible to solve for this equation form by using quadratic formula as shown below (see Figure 6).

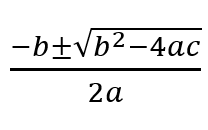
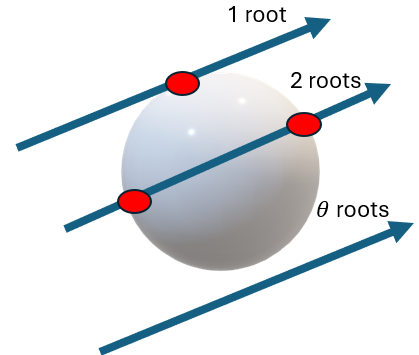
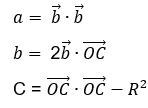


Figure 6

From the quadratic formula, we can induce the discriminant equation using the given values a, b, c. means two real solutions, means one real solution, and means no real solution. With this discriminant analysis, the identification of ray-intersection with object will be feasible. Even though ray-sphere intersection is not directly relevant to curve primitive, this process could be included in the part of ray-curve intersection later.

### Shading

Even if the primary ray intersects with anything, the basic ray tracing is not complete because there was no part for determining how these intersected surfaces interact with light. Shading and lighting play pivotal roles in transforming these intersections into visually compelling images by simulating how light behaves upon hitting objects. The aim of rendering process is to generate the appearance of something as seen from a given viewpoint. While the pixel rendering deals with visibility problem, shading is related to computing or simulating the colour of objects. Depending on the amount of light, the object’s colours, and the orientation of surfaces concerned with resources, the scene can be different. When seeing an object, the light that reaches the eyes has been reflected off its surface. This light originates from sources such as the sun, light bulbs, and flames. Once emitted, light may strike objects, which then reflect some of this light towards the viewer.

#### 1.1.4.1 Reflection & Refraction

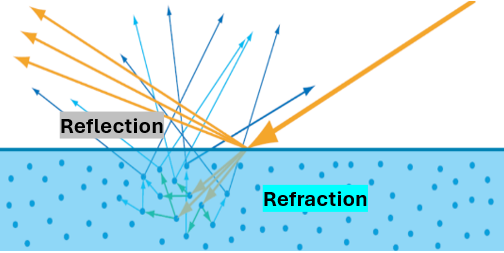


Figure 7

First, when looking at a single light particle hitting a point on the surface of a material, two primary phenomena can occur at the point of intersection ‘Reflection’ and ‘Refraction’. Figure 7 shows reflected light which bounce away from the surface and refracted light which bounce around inside the material or exit the material some distance away from where it entered.

#### 1.1.4.2 Diffuse reflection

Diffuse reflection occurs when light pass-through rough surfaces, causing the light to scatter in many directions. Unlike specular reflection, where light reflects at specific angles, diffuse one results from the microscopic variation in the surface texture.



Figure 8

#### 1.1.4.3 Lambertian reflection

In real world, there are barely perfect mirror surfaces. Most surfaces are matte, diffuse and some extent of shininess or glossiness. Unlike perfect mirrors, these surfaces do not reflect light in one concentrated direction but scatter it in random directions.

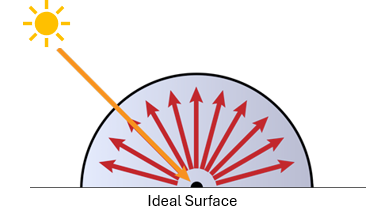


Figure 9

In ray tracing, ‘Lambertian reflection’ or ‘ideal diffuse reflection’ models this phenomenon by assuming that light is reflected uniformly in all directions from the point of incidence, irrespective of the angle of incidence. As for how the light realistically reflects off surfaces, Lambertian is suitable for natural visual effects.

**Principle**

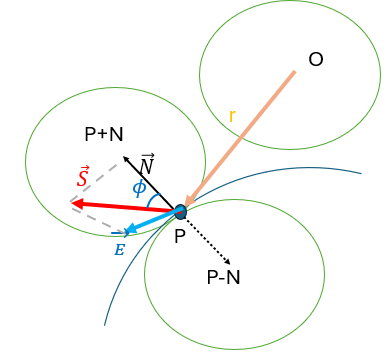


Figure 10

r : Incoming Ray

P : Surface Point

: Surface Normal

: Random point on Hemisphere

: Small random vector

: Angle Between S and N

P+N : Tangent Unit Sphere’s Centre

At the hit point P, there are one displaced in the direction of N (outside the surface) and the other in the opposite direction(-N, inside the surface). This simulation considers only the sphere on the same side as the ray origin, ensuring rays reflect outward. Interestingly, a Lambertian surface make light ray reflected proportionally to the . By the property, a reflected ray is highly likely to distribute in directions closer to the normal vector(), resulting in a non-uniform scattering. The equation 1.2 describes a random reflection off a surface.

(equation 1.2)

## Bezier Curve

Unlike simple basic primitives, the curve called ‘Bezier curve’ is flexible to handle diverse shape of objects if multiple vector points are given. It was devised by French engineer Pierre Bezier (1910–1999) who used it for designing the bodywork of cars back in 1960s. Coincidently, his curve has been playing a pivotal role in computer graphics such as user Interface design or animation. In terms of smoothness, this curve can be scaled up or down without any loss of quality.

### 1.2.1 Mathematical Concept

Bezier curves are a mathematical concept used to model smooth curves that can be scaled indefinitely. They are defined by a set of control points, within his curve's shape being determined by the relative positions of these points. The fundamental principle of Bezier curves is based on the linear interpolation of points using Bernstein polynomials, which blend the control points at varying proportions depending on a parameter ' **t** ' that ranges from 0 to 1. As ‘ **t** ’ changes, the interpolated points trace out the curve. This makes Bezier curves highly versatile and useful in computer graphics for creating intricate shapes and paths with a high degree of control and precision.

#### 1.2.1.1 Control points

When creating graphical representations of objects within a scene, it is essential to define their shapes and positions within an N-dimensional space. The characteristics of these objects—such as orientation, location, and the connections between points—are important. Take, for example, a sphere: its geometry is defined by its radius and the coordinates of its centre point in space. Similarly, in the construction of Bezier curves, we utilize a set of points known as control points.

#### 1.2.1.2 Parametric curves

In a way of description for Bezier curves, it is also called ‘Parametric curve’. Parametric curves are defined by an equation where a variable, known as a parameter, is represented by the letter 't'. This parameter 't' can range from negative to positive infinity. However, for the purposes of plotting a Bezier curve, which is only interested in the values of 't' from 0 to 1.

#### 1.2.1.3 Bernstein Polynomials

When describing parametric curves, Bernstein’s theorem of polynomials is a key component. Sergei Natanovich Bernstein devised the formula to provide a practical approach to demonstrate the Weierstrass approximation theorem [14]. Bezier curve of degree n is represented by

(equation 1.3)

The coefficient, , stands for the control points and the Bernstein basis polynomial determine the shape of the curve.

(equation 1.4)

Where the terms are called binomial coefficients. They can be obtained using factorials (“ ! ”) with the following equation:

= (equation 1.5)

For a given degree n, there are n+1 Bernstein polynomials. it is also possible to change the value of n which is the largest degree of any one term in that. Figure 7 illustrates Bezier curves of different degrees through a process of linear interpolations.

A diagram of a mathematical equation

Description automatically generated

Figure 7

Once looking at the cubic curve, it consists of connections with multiple linear interpolations(see Figure 8).

A black text with red line

Description automatically generated

Figure 8

Namely, the shape of the curve is drawn by the combining the control points weighted by scaler coefficients.

#### 1.2.1.4 The De Casteljau's Algorithm

Through the 1.2.1 parts, we know each point on a Bezier curve can be expressed as the weighted sum of the control points, using Bernstein basis polynomials. The De Casteljau’s algorithm provides the practical method for these calculations unlike the mathematical description Bernstein polynomials represented. The below example will be helpful to understand so that Its operation is rather simple (see Figure 9).

1. Begins with 4 control points P1~P4 (set initial control points).

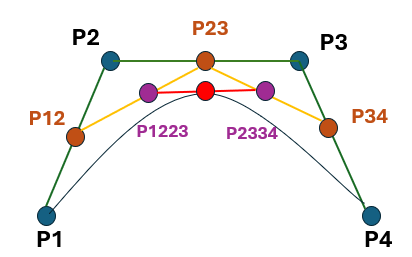


Figure 9

1. Find points on the line segments between control points (p12, P23, and P34) corresponding to a **t** value.
2. Use these new points to form new line segments (yellow lines).
3. Find the points (P1223, P2334) corresponding to **t** on these.
4. Repeat this process until only one point remains (the last point = the point on the curve at **t**).

In fact, The De casteljau’s work is numerically more stable than Bernstein’s theorem because of the property with recursive execution reduces the input errors to be negligible in the final output [9].

### 1.2.2 Bezier Surface (Mesh modelling)

For rendering dynamic shapes unlike ray-tracing basic primitives, the Bezier surface patch formed by bezier curve at 1.2.1 is one of them. It can be formed by moving curves through space and is finally defined by a grid of control points. In the mathematical representation of bezier surface point, the bezier curve equation 1.3 is utilised (see equation 1.6).

, 0, v1 (equation 1.6)

As for finding a point on the surface, this equation 1.6 shows a double sum of control points and coefficients which forms a sum of bezier curves.

As the m, n values go by, it is possible to create bi-cubic, bi-quadratic, and bi-n surface can be calculated and created based on the two parameters, named **u** and **v** (see Figure 10). However, the amount of required computation should be considered. In this project, the bicubic Bezier surface will be focused on.

Figure 11 illustrates the bicubic bezier surface for which total 4x4 control points(i.e. n=3 and m=3 in equation 1.6).

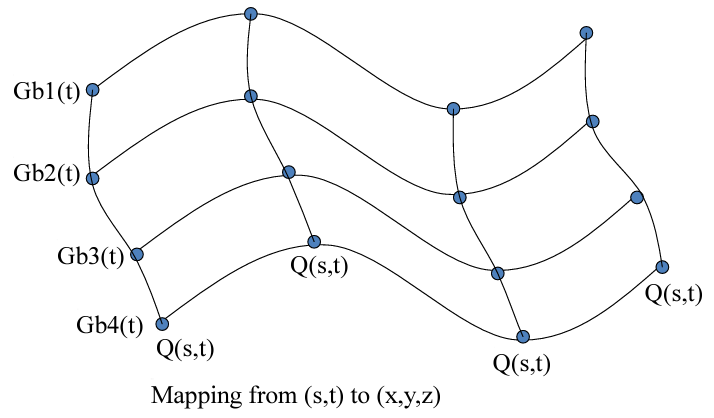


Figure 11

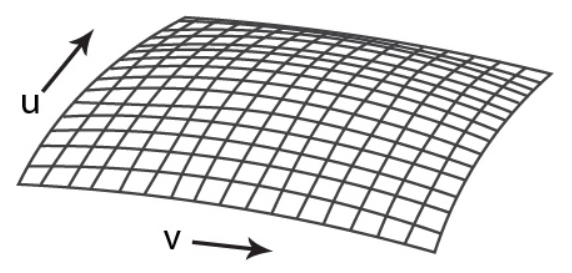


Figure 10

When it comes to the Bernstein polynomials representation (see 1.3 equation) on bicubic surface, it can be written like below.

,

# Chapter 2

### Methods

## 2.1 Development tools

To develop a Ray-tracer in which Bezier curve & surface can be rendered, the foundation of this project bases on Peter Shirley’s week series because his book is well defined to implement the ray tracing technique from scratch [12].

The IDE(or text editor) used was Visual studio because it is suitable for debugging and testing C++ application in windows platform. The reason why C++ was selected among programming languages is to provide high performance on graphics with diverse graphics libraries like OpenGL, Vulkan and so on and to give me familiarity as computer graphic module last semester used it. To build and manage Visual Studio solution file(.sln), the open-source build system ‘Cmake’ was employed. As for version control, GitHub was used to track changes to code through Git and to host code.

## 2.2 Ray-tracer construction

Many ray tracing resources are designed to use several graphic API like OpenGL or Vulkan for high performance but, this project aims to research how the ray-tracer operates and renders shape based on curve. So, generating decent image with no API would be academically good approach in this project. The architecture of the project is basically designed to be modular and extensible, namely it bases on object-oriented programming. Let it be divided into key components: ray, camera, intersection, shading system and lighting. These will be connected to make a single ray tracer through 2.2.1.

### 2.2.1 Rendering pipeline

Prior to examining the technical methods, this section describes the sequential stages of the rendering process, which is pivotal transforming a 3D scene into a visually accurate 2D image. To facilitate efficient project execution, the flow chart below provides immediate visual identification of the ray-tracer’s components (see Figure 12).

A diagram of a flowchart

Description automatically generated

Figure 12

### 2.2.2 Rendering an image

The basic principle of ray tracer consists of ray generation, intersection, and shading. This section will detail how these are implemented and how to construct a proper ray tracing environment with ancillary elements such as PPM image format or primitive templates.

#### 2.2.2.1 Ray generation



Figure 13

To represent a ray as well as the primary ray generated from camera, it is necessary to define ray class. Its way of implementation likewise follows the initial description in 1.1.1.1 Ray. Figure 13 pseudo code will be used in any circumstance relate to ray.

#### 2.2.2.2 Record for ray-intersections

Moreover, once a ray generated from camera reached any surface, ray-tracer requires the records of the ray’s information which contains intersection point, parameter value t at that point, and normal vector. Figure 14 shows the ‘Hit\_record’ class records what ray information when the ray hits a surface in the 3d space.

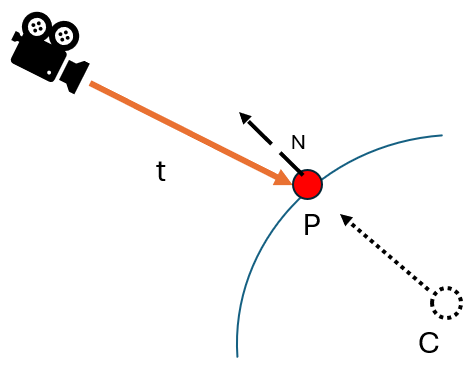
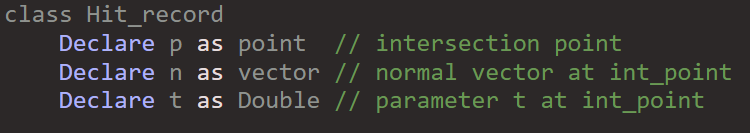


Figure 14

#### 2.2.2.3 Test for intersection

When it comes to intersection ray with any object, there is a wide variety of methods to choose from because each shape has their own geometric properties. The simplest method involves working with sphere geometry, as its foundational principles have already been outlined in section 1.1.1.3 on intersections. However, this approach can also be adapted for rendering curves later.

To determine if a ray meets with sphere or not, an algorithm for testing any intersection is needed. Figure 15 pseudocode outlines the algorithm for determining whether a camera ray intersects with a sphere in a 3D space. Firstly, it begins with defining variables which are coefficients ‘a’, ‘b’, ‘c’ and ‘oc’ derived from the mathematical sphere equation and the ray’s properties in section 1.1.1.3. And then it uses the quadratic formula to solve for ‘t1’ and ‘t2’ which stand for the scalar values along the ray’s direction vector at which the ray intersects the sphere’s surface (see figure 16). Lastly, the result should be updated in ‘hit record‘ (derived from figure 14) when hitting any surfaces. Additionally, the scalar values t1 and t2 are vary based on the camera's position, which will also be addressed in camera model part later.

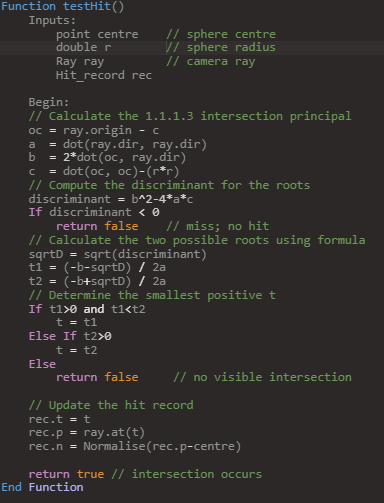


Figure 15

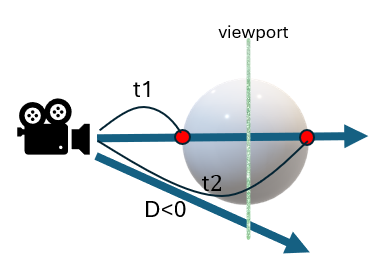


Figure 16

**2.2.2.4 Template for geometry**

The principal benefit of Object-Oriented Programming(OOP) design is the simplification of code structure. Thanks to the inheritance mechanism of C++, a plate class embodies this concept as a template for various geometric forms. All shapes in 3d space are under the range of ray hit. Therefore, ray-intersection method must be implemented in each geometry class. Figure 17 shows How an abstract function which is vacant is implemented differently by the derived class. By utilizing this mechanism, it also provides convenience in rendering multiple geometries within the camera's scene. The ‘Hittable\_list’ class, as illustrated in figure 18, acts as a container that simplifies the process of managing as well as rendering the geometries. Particularly, ‘testHit’ function iterates through each object in the list and checks for intersections to detect an intersection. According to P. Shirely’s book [12], this structure is not only efficient but also scalable, making it easy to add or remove objects dynamically without the need for individual intersection logic for each object.

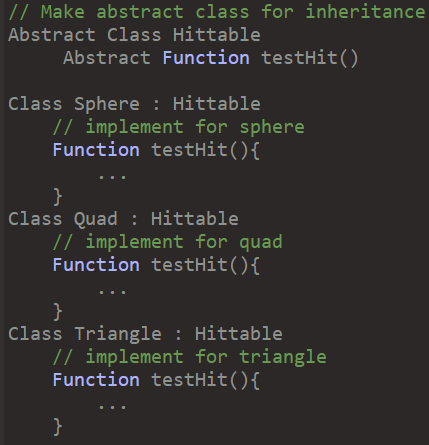


Figure 17

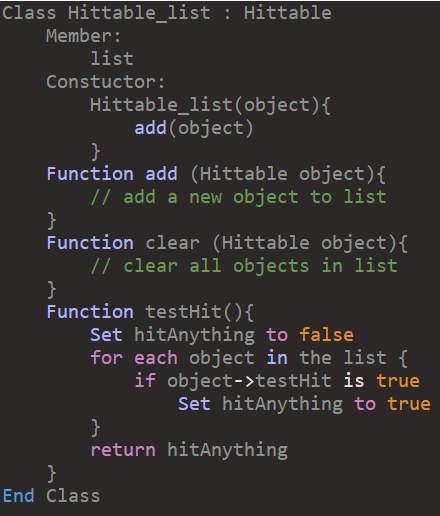


Figure 18

#### 2.2.2.5 Lighting and shading

When ray hits a surface, lighting/shading should be performed, determining what colour or light should be exhibited at this location. As described at 1.1.4, the diffuse reflection is crucial for achieving realistic shading and it can be emulated by Lambertian. This part will delve into the Lambertian material, which simulates diffuse reflection for a matte finish, and explore light emitting material.

#### Lambertian

In the same manner as above, ‘Lambertian’ class also inherits from the Material class which consists of ‘scatter’ and ‘emitted’ functions. Since it does not emit any light, only the ‘scatter’ method has been implemented in Figure 19. The scatter function generates scattering ray produced by equation 1.2 diffuse reflection. As all know, incoming ray is not required to implement scattering because this type is independent of the viewport or the direction from which light arrives

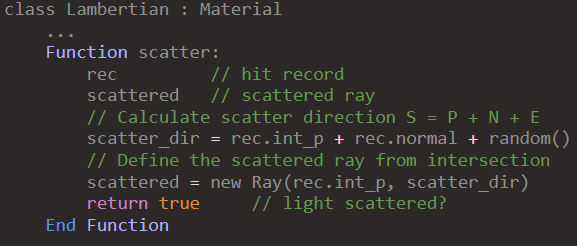


Figure 19

#### Lighting

To emulate light sources, emissive object can be used in ray tracer. Unlike the previous method, the ‘emitted’ function only operates as emitting material since it performs no reflection. Figure 20 shows that how luminant material is defined for lights. In this pseudo code, there is a texture coordinates defined by u, v values, similar with how 2D Cartesian coordinates are specified by x, y coordinates. This system allows the texture to be accurately projected onto 3D surfaces, ensuring that every point on the surface maps to a specific point on the texture(see Figure 21).



Figure 20

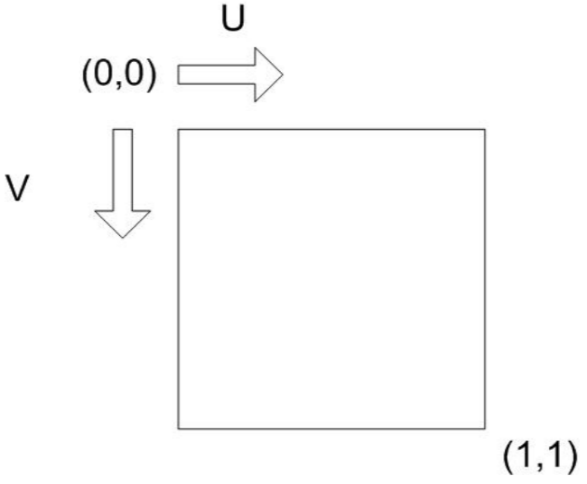


Figure 21

#### 2.2.2.6 Image format PPM

Essentially, the image data is written to ‘std::cout’, which is C++’s standard output stream, and formatted specifically for the PPM image file format. Put simply, the format begins with a header that specifies the image encoding, the dimensions of the image in pixels, and the maximum colour value for each pixel. In the case of this ray-tracer, the identifier at first line is ‘P3’, which represents the PPM plain text format, followed by the image’s size, and ‘255’ to indicate the colour value for each colour channel(red, green, and blue) [2]. Figure 22 shows the PPM format’s simplicity. On the left, there is an example of one snippet of a PPM file’s contents. On the right side, a graphical depiction of how pixels are structured in the image grid.

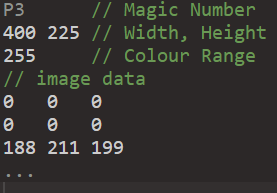
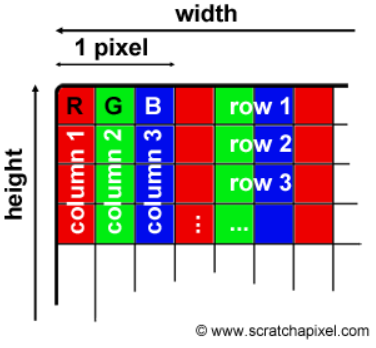


Figure 22



## 2.3 Bezier processing integration

This section delves into rather intricate process of incorporating Bezier curve functionality into ray-tracer. By translating the mathematical theory of Bezier curves into new geometry of ray-tracer, it is great step to understand bridging abstract concepts with tangible implementation. Furthermore, the practical aspect of utilizing these curves can help to explore mesh-like shape which is foundational for complex objects.

### 2.3.1 Integration ray-tracer with Bezier curve

Once the translation work completed, Bezier curve should be passed through intersection test as well because the curve will be regarded as a ray-tracer primitive. Its implementation is basically divided into direct approach or indirect approach.

#### 2.3.1.1 Design

As for the direct approach, rays are probably tested for intersection precisely with the mathematical representation without any intermediate simplification because it calculates the exact points where a ray intersects a curve. While this method might provide an accurate result by computing the actual intersection points between the ray and the curve, it is computationally more intensive than the indirect approach(see Figure 23).

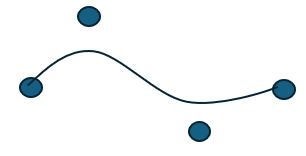


Figure 23

Thus, the indirect approach is generally preferred due to its efficiency for computation. Unlike previous method, it simplifies intersection testing for Beizer curves by approximating the curve with a sequence of geometric shapes. This process, often referred to as ‘tessellation’, involves breaking down the curve into a chain of simpler elements like triangles(see Figure 24).

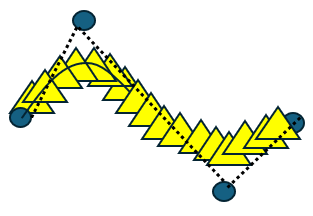


Figure 24

By tessellating the curve into basic primitives, it transforms the complex problem of finding intersections with a non-linear shape into a series of straightforward intersection tests with these simpler shapes.

#### 2.3.1.2 Example with spheres

Here is an example that demonstrates this method in action, utilizing a series of spheres to approximate and perform intersection tests on a Bezier curve (see Figure 25). Each sphere along the curve acts as an individual test unit. If a ray intersects any of these spheres, it suggests a potential intersection with the actual curve segment that the sphere approximates.

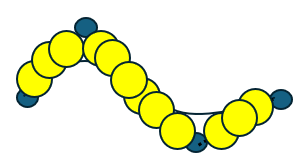


Figure 25

On the left of Figure 26, it indicates how the input data are added in a list for control points. Even though four points are given for cubic curve, but also different number of points can be added for diverse shape of curve. On the right, a test intersection method is shown which is slightly different with sphere’s test described at 2.22 test for intersection. Instead of a single sphere’s data, curve points are used, replacing centre point of curve point in a loop. The logic is basically straightforward because it refers to the property of existing geometries (see figure 27).

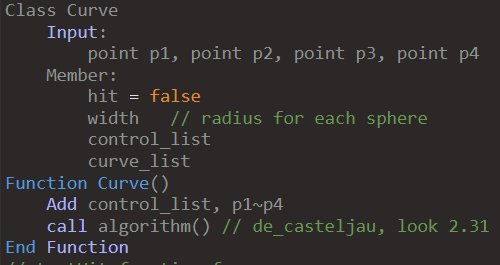


Figure 26

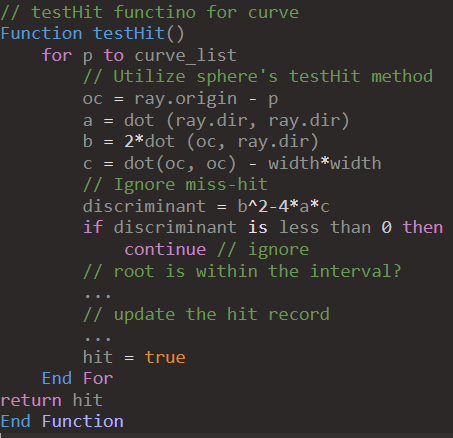


Figure 27

### 2.3.2 Fetch Implementation using curves

Having outlined the foundational methods for Bezier curves, it’s time to deal with the application and adaptation of these principles to Bezier surface. To recapitulate the content of section 1.2.2, it is formed by a network of Bezier curves controlled through a grid of control points and Each curve in this grid contributes to the final shape of the surface.

#### 2.3.2.1 Point on Bezier surface

For this project, the Bezier surface will be developed by translating these curve-based concepts into a 2D mesh that is what this section aims to. The process can be usually accomplished through tessellating the surface, which subdivides the continuous surface into discrete triangular elements. The reason why this approach is used is that the elements are easier to manage computationally and are well suited for rendering and intersection calculations in a ray-tracing environment. To perform the tessellation, it is necessary to compute points along the surface. The below equation 1.7, which indicates a point on the surface, has already been introduced in 1.2.2 Bezier Surface section.

0, v1 (equation 1.7)

By leveraging above equation, the method called “surfacePoint” can be defined(Figure 28).

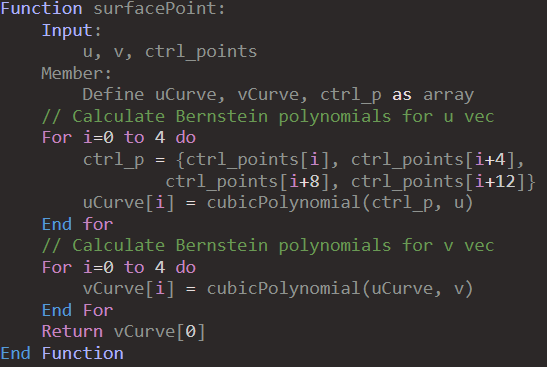
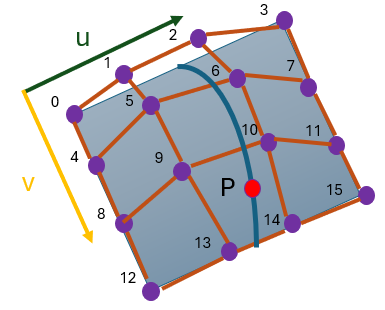


Figure 28

From the 4x4 control points grid, a parameter ‘u’ is used to evaluate a position in 3D space along each curve, resulting in four new control points. Through the calculated control points, it can define an auxiliary curve based on the v parameter. This secondary curve represents a specific point on the Bezier surface for the given ‘u’ and ‘v’ parameter pair, facilitating the tessellation of the Bezier surface. When discussing the ‘cubicPolynomial’ method, it’s important to note that this method is designed specifically for cubic curves, meaning it is optimized for curves of degree three. The equation 1.8 below aligns closely with Bernstein polynomials described at 1.2.1.3 Bernstein polynomial.

(equation 1.8)

To translate this polynomial into code, it can be expressed simply in Figure 29. This function operates by determining the influence of each control point on the curve at the specific instance defined by ‘ **t** ’ which is a parameter in the range [0,1] for both ‘**u**’ and ‘**v**’ directions. Within the degree loop, the ‘Bernstein’ function calculates its coefficients, which are used to weight the control points’ contribution to the curve. And then, the result accumulates these weighted control points to find the point on the curve.

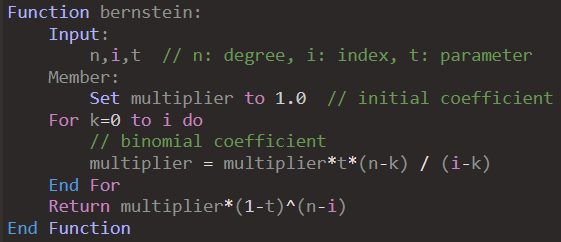


Figure 30

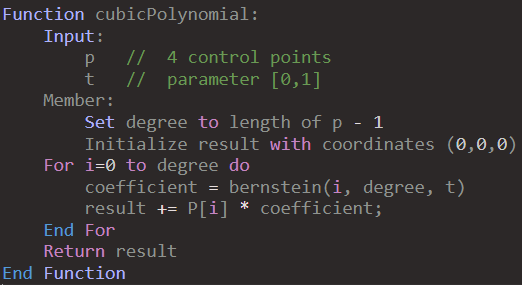


Figure 29

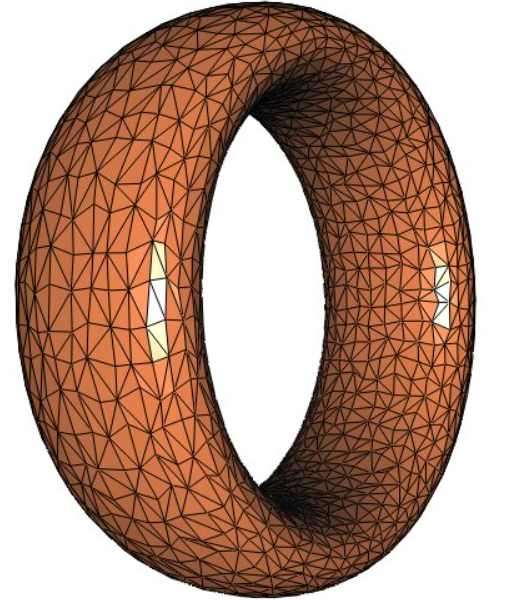
Figure 30 shows how the ‘Bernstein’ equation below translates into a function in code. The point in the code is to compute binomial coefficient, which is also studied at section 1.2.1.3. However, this code does not calculate the binomial coefficient directly as 1.4 equation.

Instead, can be expanded into a product of fractions:

(equation 1.9)

On each iteration, the term ‘(n-k)’ from the numerator follows the descending pattern ‘n, n-1, …, n-i+1’. While the term ‘(i-k)’ from the denominator is follows the sequence ‘i, i+1, …, 1’ in reverse. The calculation ‘multiplier \* (n - k) / (i - k)’ for each ‘k’ constructs the binomial coefficient by multiplying the current multiplier the corresponding fraction of the sequence terms.

#### 2.3.2.2 Tessellation over surface

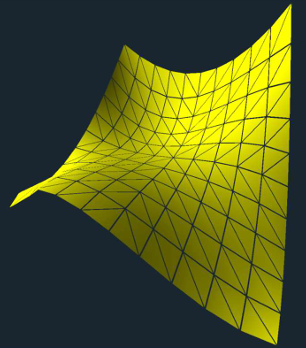


**Figure 31**

Tessellation is the process of dividing the surface into smaller geometric shapes that can be efficiently rendered by the GPU. In a similar way to how a Bezier curve is made up of basic primitives, the surface can be broken down into polygons which are then used to create a detailed approximation of the original shape.

#### 2.3.2.3 Design

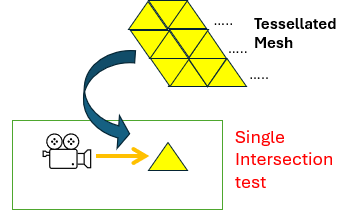
Among them, Triangle is the simplest polygons, which makes them highly efficient for GPU processing. Therefore, this project plans to use a tessellation approach that divides the surface into a mesh of triangles. The algorithm first establishes a grid of points over the surface, creating a series of quads. Each quad is then split into two triangles, ensuring that every edge aligns precisely without overlapping or gaps.



**Figure 32**

#### 2.3.2.4 Integration

Once the mesh are divided into multiple triangles, it’s time to test ray-intersection between camera ray and surface. In terms of efficiency and accurancy, testing the entire mesh at once can complicate the process of determining which triangle intersects closest with the ray in complex scenes. This method may occur inclreased computational overhead and reduced precision in the rendering results. Therefore, each tessellated triangle will be individually tested.



**Figure 33**

# Chapter 3 Results and Improvement

Finally, the result of image rendering by ray tracer will be displayed and evaluated. This chapter includes screenshots what the project produces with comment and discussion for improvement of image performance.

## 3.1 Test with Bezier curve modelling

|  |  |  |  |
| --- | --- | --- | --- |
|  | Sphere chain tube | Triangle chain tube | Quadrangle chain tube |
| Result | A green circle with black dots  Description automatically generated | A green spiral in the air  Description automatically generated | **A green arrow on a blue square  Description automatically generated with medium confidence** |

When it comes to testing the Bezier curve, the ray-tracer operates correctly and renders the curve as expected. This curve model can utilize the basic primitives such as spheres, triangles, and quadrangle, making a tube along the curve for efficient rendering in a ray tracing environment. That chain approach is often called ‘Sweeping’ or ‘extrusion’ in CAD(computer-aided design). Table 3.1 displays the results corresponding to each primitive type.

Table 3.1: Bezier curve with different types of basic primitives

## 3.2 Improve image performance

In terms of image performance shown in table 3.1, it seems to need improvement in rendering algorithm. Therefore, several ways for enhancement will be explored.

### 3.2.1 Aliasing

When an image rendered using pixels, a phenomenon looks like staircase or irregular colour pattern commonly occurs on the screen. Such effects are called “Aliasing” (see figure 34). If the sampling rate(the number of rays cast) during ray tracing is not high enough, details in the scene may not be properly represented, resulting in aliasing.



Figure 34

#### 3.2.1.1 Anti-aliasing

For reducing these, Anti-aliasing(AA) algorithms have been used in computer graphics. Among various ways of AA, Super sample Antialiasing(SSAA) will be applied on rendering algorithm. According to research [11], SSAA is described as the simplest Anti-Aliasing technique, where multiple scene samples are taken at different locations relative to a screen pixel and then computationally combined to produce an anti-aliased image.

|  |  |
| --- | --- |
| **Tracing a ray per each pixel** | **Multiple rays within a pixel** |
|  |  |

Table 3.2 : Comparison between original ray-cast and SSAA

#### 3.2.1.2 Improvement

As compared to Figure ?, the result by above technique clearly shows the difference of image quality in table 3.2.1. When zooming further in, the staircasing problem which has been slightly blurred can be captured. However, there are still many black dots spread out in the scene.

|  |  |
| --- | --- |
| A green leaf with white background  Description automatically generated | A green spiral on a blue surface  Description automatically generated |
| Zoom in | Zoom out |

Table 3.2.1: SSAA technique applied Bezier curve image

### 3.2.2 Shadow acne

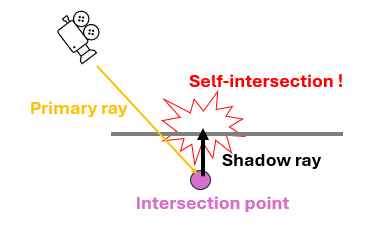


Figure 35

By the finite precision of floating-point numbers in computer graphics, a subtle bug, which is called ‘shadow acne’, happens frequently. This self-intersection in figure 35 is mostly caused because the ray intersects a point slightly below the surface of the object, leading to erroneous calculations for shadows.

#### 3.2.2.1 Shadow bias

To cover these dots problem in the scene, a small offset known as ‘shadow bias’ is introduced in the shadow computation process. This offset can be implemented in two ways. one involves setting minimal threshold number on **t** value, which indicates the ray parameter introduced in Section 1.1.1 on Ray. In other words, this threshold ensures that the ray does not start too close to the actual surface, thus preventing the ray from self-intersecting. The other method involves adding a bias value directly at the shadow point in the way of to avoid self-intersection.

|  |  |  |
| --- | --- | --- |
| Method | Ignore small ‘t’ value | Displace shadow point |
| Visual |  | A diagram of a normal and displeased line  Description automatically generated |
| Description | Set minimal threshold on ‘t’ | Add bias value in N vector direction |

Table 3.2.2.1 : Two shadow bias methods

#### 3.2.2.2 Improved result

Table 3.2.2.2 displays the scene of a Bezier curve without ‘shadow acne’ under different lighting conditions. On the left, the background colour without illumination. On the right, lighting is introduced, which highlights the textural details of the curve, yet acne patterns still occur due to computational error in rendering process.

|  |  |
| --- | --- |
| A green spiral on a white surface  Description automatically generated | A green object in the air  Description automatically generated |
| Turn off the light in background color | Turn on the light in dark |

Table 3.2.2.2 : Improved two images via series of above processes

## 3.3 Test with Bezier Surface modelling

In succession of curve rendering, the extension to Bezier surface modelling presented a challenge that builds on the principles of curve manipulation. This testing section will explore whether the process in ray-tracing space can successfully achieve the objectives intended for this project. Table 3.3 demonstrates the final output rendered with two different lighting conditions.

|  |  |
| --- | --- |
| A red paper fish on a blue surface  Description automatically generated | A red hat in the air  Description automatically generated |
| Turn off the light in background color | Turn on the light in dark |

Table 3.3 : the results of bezier surface

## 3.4 Model evaluation

### 3.4.1 Rendering time

Typically, ray tracing takes longer to render compared to other rendering techniques like rasterization, which processes images more rapidly by converting 3D models into 2D pixel representations using a mesh of triangles [B. Caulfield, NVIDIA]. Table 3.4.1 shows that how long each of proposed primitives takes to render in ray tracer space.

|  |  |
| --- | --- |
| Type | Rendering Time with light |
| Bezier Curve with spheres | 3min 4sec |
| Bezier Curve with triangles | 55min 8sec |
| Bezier Curve with quadrangle | 43min 58sec |
| Bezier Surface | 7min 8 sec |

Table 3.4.1 : Analysis of computation time for each type

### 3.4.2 Analysis

As mentioned in section 3.4.1, the rendering times for different bezier primitives vary significantly. The Bezier surface, with the longest rendering time, suggests that surfaces require more computational resources compared to simpler forms like spheres.

In particular, the rendering performance of Bezier curve with triangles and quadrangles was markedly worse compared to spheres. The intersection tests for spheres involve simple geometric calculations, allowing for efficient rendering. In contrast, both triangle and quad require turning curves into render-able objects, namely tessellation into triangles or quadrangles) might be inefficient. If the curves are subdivided into an unnecessarily high number of smaller segments, this could increase rendering time.

In terms of looking at computational algorithm, while sphere uses simple arithmetic to find the intersection point, triangle checks if the point lies inside the triangle using methods like calculating the area of sub-triangles formed with the intersection point and comparting it with the total area. Quad also requires complex geometric calculations to determine not just an intersection with a plane, but also precise positioning within bounded quadrangle space.

Given these observations, there are several strategies that could be considered to optimize rendering times.

### 3.4.3 Plans for its solution

#### 3.4.3 .1 Efficient data structure - BVH

the main constraint in ray tracing is the quantity of ray-object intersections, with the rendering time increasing proportionally to the number of objects [13]. This is where BVH (Bounding Volume Hierarchy), comes into play. The term “Bounding Volume(BV)” refers to any shape that can encapsulate an object. The fundamental concept is to simplify object interactions by using BV or spatial decompositions thus reducing the frequency of intersection checks between objects [6]. By organizing them in a hierarchically tree, it boosts computation, dismissing large areas of non-intersection.

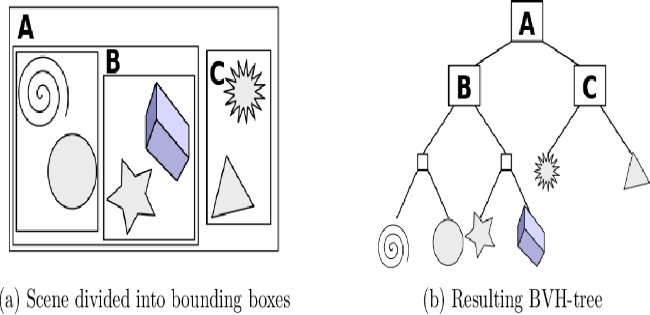


Figure 36

#### 3.4.3 .2 Parallel processing

In 3D graphics rendering, parallel processing, known as distributed workload design, can provide an effective solution for reducing computational burdens. As previously discussed, rendering a single pixel requires massive calculations involving colour, light intensity, direction, and surface properties. Through parallel processing, these calculations can be accelerated, dividing into subtasks, and distributing across multiple CPU or GPU cores. When it comes to the computation of normal vectors for a triangular patch, parallel processing improved the speed of performance compared to traditional iterative methods [5].

# Chapter 4 Discussion

This upcoming section will address unexplored areas, including limitations, conclusions, and directions for future work.

## 4.1 Limitations

The output of this project is limited to a PPM file image, which lacks a user interface within the window. This format namely restricts the ability to interact dynamically with the rendered results. Moreover, this ray tracer requires considerable computational effort, as each ray must be checked for intersections against potentially complex bezier patches, which can require significant processing power and time.

Structurally, the design and implementation of curve and surface are hardcoded. In other words, implementation of surface is designed to work with a specific number of control points. If the input data deviates from this expectation (specifically, more than the expected 16 points for a Bezier surface), it may not function correctly.

## 4.2 Conclusions

In conclusion, this project explored the interaction of complex curve shapes, specifically Bezier surfaces, within a ray tracing environment. These interactions are rather more complicated than generic geometries, presenting new challenges in terms of computational demands and rendering precision.

Although this project deals with especially Bezier curves among various types such as B-splines, NURBS(Non-Uniform Rational B-splines), and so on, it was discovered that this curve offers a simplicity, making them particularly suited for applications where precise shape manipulation is required. The flexibility and ease of use of Bezier curves have allowed for detailed and accurate modelling of surface.

From the integration of curve primitives and ray tracing, mathematical principles are properly applied to render surfaces derived from Bezier curves, enabling the depiction of various complex shapes seen in CG movies and animations. Bezier curves, which are generated using control points, offer a robust way to model smooth and scalable surfaces essential for creating visually compelling graphics.

Overall, the performance of images rendered in the no lighting condition works properly. However, in situations involving illumination, there is still some noticeable noise. This indicates a need for further optimization in the ray tracing algorithms to image quality and reduce artifacts, particularly in scenarios with lighting conditions. As for computation time about the process, it is significantly lengthy, highlighting the need to improve its efficiency.

## 4.3 Ideas for future work

### 4.3.1 Real-time Ray tracing

What if a surface moves around and the rendering responds in real-time to user inputs or environmental changes? When it comes to video games or simulations where scenes must update instantly in response to user inputs, the current offline rendering, which may take hours to render for a single frame high-quality output, is unsuitable. With the development of graphic cards and APIs like Microsoft’s DirectX Raytracing, the technology known as ‘Real-time ray tracing’ is now used in mainstream applications [4]. Delving deeper into dynamic rendering, this tracing method proves to be a fascinating study as it processes frames in milliseconds to maintain smooth motion.

### 4.3.2 Path tracing

While Ray tracing traces the path of light rays primarily to compute their direct interactions with objects, Path tracing continues to trace the path of rays after their first intersection with surfaces, accounting for a series of potential subsequent scatterings. This advanced capability allows path tracing to create highly realistic images and more naturally simulate complex optical effects like global illumination and caustic effects, surpassing the capabilities of ray tracing. which requires additional algorithms to replicate these effects [1]. If this project is extended to include path tracing, it would give valuable insight into how it produces the enhanced realism in comparison to ray tracing.

# List of References

*<It is expected that the list would reflect the breadth and depth of scholarly research undertaken by the student during the course of the project.>*

[1] B. Caulfield at NVIDIA, 2018, What’s the Difference Between Ray Tracing and Rasterization?, <https://blogs.nvidia.com/blog/whats-difference-between-ray-tracing-rasterization/>

[2] Clemson University, Simple Image File Formats , chrome extension://efaidnbmnnnibpcajpcglclefindmkaj/https://people.computing.clemson.edu/~dhouse/courses/405/notes/ppm-files.pdf

[3] D. Meister et al., 2021, A Survey on Bounding Volume Hierarchies for Ray Tracing, Computer graphic forum, Vol: 40, Issue: 2, DOI: <https://doi.org/10.1111/cgf.142662>

[4] Epic Records at Unreal Engine, <https://www.unrealengine.com/en-US/explainers/ray-tracing/what-is-real-time-ray-tracing>

[5] G. Zhang and F. Ye, 2022, 3D Complex Scene Rendering Acceleration Method Based on Parallel Processing*, IPEC '22: Proceedings of the 3rd Asia-Pacific Conference on Image Processing, Electronics and Computers*, Pages 370–374 ,DOI: <https://doi.org/10.1145/3544109.3544176>

[6] J.T. Klosowski et al., 1998, Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs, *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS*, Vol: 4, Issue 1, DOI: [10.1109/2945.675649](https://doi.org/10.1109/2945.675649)

[7] J. T. Whitted at NVIDA, 2018, Why Ray Tracing Is a Natural Choice for Global Illumination, *A Ray-Tracing Pioneer Explains How He Stumbled into Global Illumination*, <https://blogs.nvidia.com/blog/ray-tracing-global-illumination-turner-whitted/>

[8] K. Nakamaru and Y. Ohno, 2002, Ray Tracing for Curves Primitive, *International Conference in Central Europe on Computer Graphics and Visualization*, <https://www.semanticscholar.org/paper/Ray-Tracing-for-Curves-Primitive-Nakamaru-Ohno/2d279bb6f6ab1f034659c213e627d19d0c8fb6f9#paper-topics>

[9] M. Hanik and E. Nava-Yazdani, 2024, De Casteljau’s Algorithm in Geometric Data Analysis: Theory and Application, Vol: 110, DOI: <https://doi.org/10.1016/j.cagd.2024.102288>

[10] NVIDIA, Ray Tracing, <https://developer.nvidia.com/discover/ray-tracing>

[11] N. DAMERA-VENKATA and N. L. CHANG, 2009, Display Super-sampling*, ACM Transactions on Graphics*, Vol: 28, Issue 1, Article NO: 9, pp 1~19, DOI : <http://doi.acm.org/10.1145/1477926.1477935>

[12] P. Shirley, January 2016, Ray Tracing in One Weekend, chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.realtimerendering.com/raytracing/Ray%20Tracing%20in%20a%20Weekend.pdf

[13] P. Shirley, March 2016, Ray Tracing: The Next Week, chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.realtimerendering.com/raytracing/Ray%20Tracing\_%20The%20Next%20Week.pdf

[14] R. T. Farouki, 2012, The Bernstein polynomial basis: a centennial retrospective, VOI: 29, *Computer Aided Geometric Design*, Issue 6, pp: 379-419, DOI: <https://doi.org/10.1016/j.cagd.2012.03.001>

**<Image Reference>**

[Figure 1] <https://developer.nvidia.com/discover/ray-tracing>

[Figure 2] <https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays/definition-ray.html>

[Figure 3] <https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-generating-camera-rays/generating-camera-rays.html>

[Figure 11] chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.cs.colostate.edu/~cs410/yr2016fa/more\_progress/cs410\_F16\_Lecture21.pdf

[Figure 21] <https://learn.microsoft.com/en-us/windows/win32/direct3d9/texture-coordinates>

[Figure 22] <https://www.scratchapixel.com/lessons/digital-imaging/simple-image-manipulations/reading-writing-images.html>

[Figure 31] <https://www.researchgate.net/publication/262271082_Curvature_tensor_computation_by_piecewise_surface_interpolation/figures?lo=1>

[Figure 32] chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://web.engr.oregonstate.edu/~mjb/cs519/Handouts/tessellation.1pp.pdf

[Figure 36] [https://www.semanticscholar.org/paper/Parallelization-of-BVH-and-BSP-on-the-GPU-Imre/961ca2300140ec357b8f878a534858b09bb84av v vbn v vvbn v cvbn vbn dcxgfvbn vdcxgfvbn vvbn vdcfvbn vdcxgfvbn v v bc/figure/6](https://www.semanticscholar.org/paper/Parallelization-of-BVH-and-BSP-on-the-GPU-Imre/961ca2300140ec357b8f878a534858b09bb84av%20%20v%20%20vbn%20%20%20v%20vvbn%20v%20cvbn%20vbn%20dcxgfvbn%20%20vdcxgfvbn%20%20vvbn%20%20vdcfvbn%20%20vdcxgfvbn%20%20v%20v%20%20bc/figure/6)

# Appendix A Self-appraisal

## A.1 Critical self-evaluation

Even though the project went as intended and demonstrated the core capabilities of this project’s ray-tracer, it may have missed opportunities to explore more complex aspects of realistic rendering.

Firstly, it’s unfortunate that the analysis focused primarily on curve modelling with spheres. While assessing the performance differences among various geometric representations of curves, I was so concerned that whether the focus might lead the project beyond its intended scope.

Secondly, the ray-tracer is fundamentally specialized for realistic rendering, particularly in terms of light interactions. Therefore, it might have been more beneficial for this project to focus on mechanisms such as how texturing on surfaces enhances realism or how the reflection and refraction of light affect image quality, depending on the material properties.

When it comes to a critical point of concern in this project, it is the prolonged rendering times, particularly when constructing curves with triangles and quadrangles. Typically, surface rendering involves more complex calculations due to the higher dimensionality and additional detail required for accurately depicting 3D surfaces. But the result is completely different. I think it might be inefficiencies in the approach.

## A.2 Personal reﬂection and lessons learned

Through the course of this project, an understanding was developed regarding the operation and principles of ray tracing, especially in the context of RTX graphic cards of NVIDIA. Initially, the specifics of how ray tracing functions on such advanced hardware were not well understood. But the project facilitated a comprehensive learning experience, illuminating the sophisticated mechanisms that enable ray-tracer to produce realistic images by simulating the physical behaviour of light.

To improve image quality, a primary strategy often involves increasing the number of rays traced. while this approach directly enhanced the detail and accuracy of the rendered images, it also proportionately increased the computational load. This implies that achieving good performance necessitates a corresponding increase in computational capacity. I think it would be worthwhile to explore ways for reducing computation or processing time in terms of optimisation.

To conclude, I have learned that producing high-quality images is crucial, yet optimizing the rendering process to deliver these images efficiently within a shorter timeframe is equally important.

## A.3 Legal, social, ethical and professional issues

### A.3.1 Legal issues

In the context of this project on Ray tracing and curve rendering, legal issues can be relevant. Intellectual property rights are important, as the project must ensure that all utilized algorithms and software tools are either developed in-house or properly licensed. As explained at the beginning of this project, the overall ray-tracer design of this project comes from Peter Shirley’s book “Ray tracing in a weekend” [12].

### A.3.2 Social issues

When it comes to social implications, they are subtle to this project. However, the advancements in rendering technology could potentially impact on employment in creative industries by changing the skill sets required, potentially leading to job displacement or new opportunities.

### A.3.3 Ethical issues

When considering ethical issues on this project, they are not prominently applicable. This project focuses on 3D rendering performance, which involves deterministic algorithms based on physics and geometry, not involving personal data. If it targets at environment, the project’s computational demands are negligible. Thus, ethical considerations specific to data privacy or environmental impact are largely non-relevant here.

### A.3.4 Professional issues

This project does not include any collaboration with any third parties, this part would mainly indicate adherence to academic standards, research integrity and so on. To ensure academic integrity, all sources utilized in this project were accurately referenced, and appropriate acknowledgments were made to the foundational research and contributions of others. Additionally, all tools employed in the project were used in accordance with their licensing agreements to uphold professional standards of conduct.

# Appendix B External Materials

## B.1 Code design

The overall code structure derived from Peter Shirley’s ray tracing tutorial book [12].

## B.2 Tools

Draw.io was used to create the rendering pipeline [Figure 12]