VIETNAM NATIONAL UNIVERSITY OF HOCHIMINH CITY
THE INTERNATIONAL UNIVERSITY
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

RENDERING SYSTEM BASED ON RAY TRACING TECHNIQUE

By
Le Huu Sy – ITITIU19067
Pham Tran Anh Phuc - ITITIU19182
Duong Minh Nhut – ITITWE19024

A thesis submitted to the School of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
Bachelor of Information Technology/Computer Science/Computer Engineering

Ho Chi Minh City, Vietnam
Year

THE TITLE OF THE THESIS IS LOCATED HERE

APPROVED BY:

_____ ,
Nguyen Van A, Ph.D, Chair (*Example*)
*(*Type Committee names beneath lines*)

_____
(*Typed Committee name here*)

_____
(*Typed Committee name here*)

_____
(*Typed Committee name here*)

_____
(*Typed Committee name here*)

THESIS COMMITTEE
(Whichever applies)

ACKNOWLEGMENTS

With deep gratitude and respect, I acknowledge the professional guidance of Dr. Nguyen Van Sinh and Dr. Le Duy Tan. The constant support and knowledge provided by the course helped us achieve our goals.

Thank you to all members of the team for working together with all their efforts to contribute to the achievement of the project's results. Gratitude is also expressed to the members of my reading and examination committee.

Note: Paper A4, Top: 2.5cm; Bottom: 2cm; Left: 3cm; Right: 2cm

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Today, the rapid advancement of technology, especially opening up some new technology industries such as VR (virtual reality)/AR (Augmented reality), and metaverse technology pioneered by Facebook,... has opened up many opportunities and promoted graphics. Computers have become more advanced than ever. At the same time, thanks to hardware development, image processing techniques that have long been deprecated are well received, including Ray tracing. In this project, we build a rendering system of animated scenes from scratch using ray-tracing technology to capture ideas from this technique and apply the knowledge from the Computer Graphics course provided. We also optimise this technique by improving the rendering process through multithreading and octree algorithms. Techniques are deployed from constructing an object to lighting based on the Phong illumination model and casting Shadow based on principles of light: reflection and refraction. The project has rendered images at a basic level, which algorithms can optimise. The results have also shown that rendering using the ray tracing technique varies on the processor speed of the CPU of each machine difference.

CHAPTER 1

INTRODUCTION

1.1. Background

Nowadays, there are computer games and applications available that use a cluster of PCs to run in real-time raytracing[7]. The question is whether real-time ray tracing on consumer computers with a single CPU will be achievable with upcoming GPU (Graphics Processing Units) generations.

Multiple techniques and algorithms have been made to attempt to accelerate the time to the process of rendering using ray-tracing such as of Graphica Computer Corporation [8], to solve the problem of speed and aliasing by using multithreading and antialiasing, which is also the approach we choose to recreate in our project as well as the approach inspired by Whang et al of applying octree [9].

With these advances made and much more to come, it helps produce a more satisfying result of rendering in computer graphics, thus helping digital art such as 3D games, CGI, movies, etc.. have a more realistic feel.

1.2. Problem Statement

The reason for these advancements in ray-tracing attempting to improve and accelerate the time needed to render utilising ray-tracing is due the resulting rendering using ray-tracing gives the result realistic image, but its major flaw is that it takes a long time to render every pixel shown on the scene [8]

For us, we would like to take it upon ourselves and attempt to implement these approaches and understand what the core problems are and concepts in ray-tracing, in addition, to fully knowing how a renderer functions.

1.3. Scope and Objectives

Scope: Our focus is on understanding how ray-tracing functions and implementing various methods of speeding up the rendering process to completely understand the ideas, issues, and recommendations inspired by previous work.

Objective: The key objective we want to achieve is to utilise what we've learned in this Computer Graphics course and apply it to the project; thus, we decided on this project of a ray-tracing-based renderer.

1.4. Assumption and Solution

Ray tracing take a long time because there are so many intersection tests. Since each ray must be compared to every object in a scene, the main strategy for speeding up ray tracing is to lower the overall number of hit checks [7] without affecting the quality of the resulting image. Approaches to solving these problems have been made multiple times in the past, such as applying multithreading, anti-aliasing, and octree, as mentioned above. In this project, we attempt to recreate those approaches and build a renderer.

CHAPTER 2

LITURATURE REVIEW/RELATED WORK

Raytracing has been the method of choice for rendering Animated Scenes since 1968. Parker et al [18] proved that interactive framerates could be attained on a large shared memory supercomputer using a full-featured ray tracer. Nonetheless, this method is too slow and inefficient for interactive use due to limitations in terms of performance. Now with the development of technology leading to improvements in hardware that facilitates and faster algorithm execution environment, this has recently changed as the feature of raytracing. Real-time functionality, however, also creates brand-new issues not present in an offline setting. The ability to follow animated scene/animated-moving scene material interactively is notably provided by real-time ray tracing.

However, the truth is that before approaching hardware advancement to optimise raytracing, research has focused on rebuilding the spatial data structure as rendering changes due to new changes. Objects are reconstructed or updated with new parameters: position, shadows, and camera,... leading to a bottleneck in ray tracing speed. It is this problem that sets the stage for the birth of different algorithms and data structures. Studies also often focused on improving: scene/light complexity, object structure, type of motion and the coherency of the rays.

Related works include 1, ray aggregation techniques, including kd-tree tracing proposed by Wald et al.'s [2], frustum- or interval arithmetic-based techniques first presented by Dmitriev et al in the form of ray tracing on triangle intersection[19], traversal-based techniques were first presented by Reshetov et al [20] 2, Dynamic Ray techniques with typical methods such as spatial subdivision techniques approach and object hierarchies approach

2.1. Ray Aggregation Techniques

Ray aggregation techniques are based on processing aggregates of Rays into aggregates of ray beams that form packets/frustum. The ray beams form the rules for the algorithm to be optimized. With this approach, the methods all focus on dealing with the problem of how to build the structure of the kd-tree in the fastest way to scratch every frame, thus supporting arbitrary modification of the object geometry. In the work of Wald et al, kd-tree-based packets reinforce the coherence of primary and shadow rays by being approached and handled by four ray-triangle tests in parallel [2], or Liang et al [3] estimate distribution from visible primitives to estimate the distribution of rays and process rays by the stream. In circumstances where the whole packet misses the triangle, Dmitriev et al employed the bounding frustum to remove triangle intersections [19].

2.2. Dynamic Ray Techniques

The Kd-tree-based approach technique has been thoroughly investigated with optimisations and is even acknowledged as the best way for raytracing [4]; nonetheless, the update on the context is a drawback of this method when applied in the scene that is dynamic because even minor modifications to the scene's shape generally render the tree incorrect. Therefore, new methods with approaches to handle rays in complexity when objects/geometry become complex when those become dynamic. There are two ways of approaching these techniques: Spatial subdivision techniques approach with the work of Razor accepts Catmull-Clark [16] subdivision patches, d Benthin's free-from ray tracing system accepts cubic Bezier splines and Loop subdivision surfaces [17] and object hierarchies approach (so-called bounding volume hierarchy - BHV) with the work of Goldsmith and Salmon[21], who first use cost predictions to combat the addition of primitives to the tree. Muller and Fellner [22] later published research on building an object hierarchy based on function costs, identifying areas with uniformly dispersed items, and locally integrating unified spatial subdivisions into scene trees.

CHAPTER 3

METHODOLOGY

3.1. Overview

In reality, light sources emit light rays, spreading out in all directions. The light rays travel in a straight line in one medium, and when they hit objects, those rays will be reflected, and, finally, they will meet our eyes. Our eyes then can calculate the colour of those rays and rebuild the images of the object smaller in size into our brains. The way the raytracer engine works is completely opposite to the light in reality. It uses a camera which acts as our eyes, and that camera shoots view rays through pixels of the screen. When it interacts with objects, it is reflected, and those rays will continue to travel until they hit the light sources. Then the light will return its colour which will be combined with other information about the object to calculate the final colour for the pixel the ray passes through. When all the pixels are filled, we receive the scene of the objects.

3.2. System Design

3.2.1. User Interface design (GUI)

Structure of User Interface design

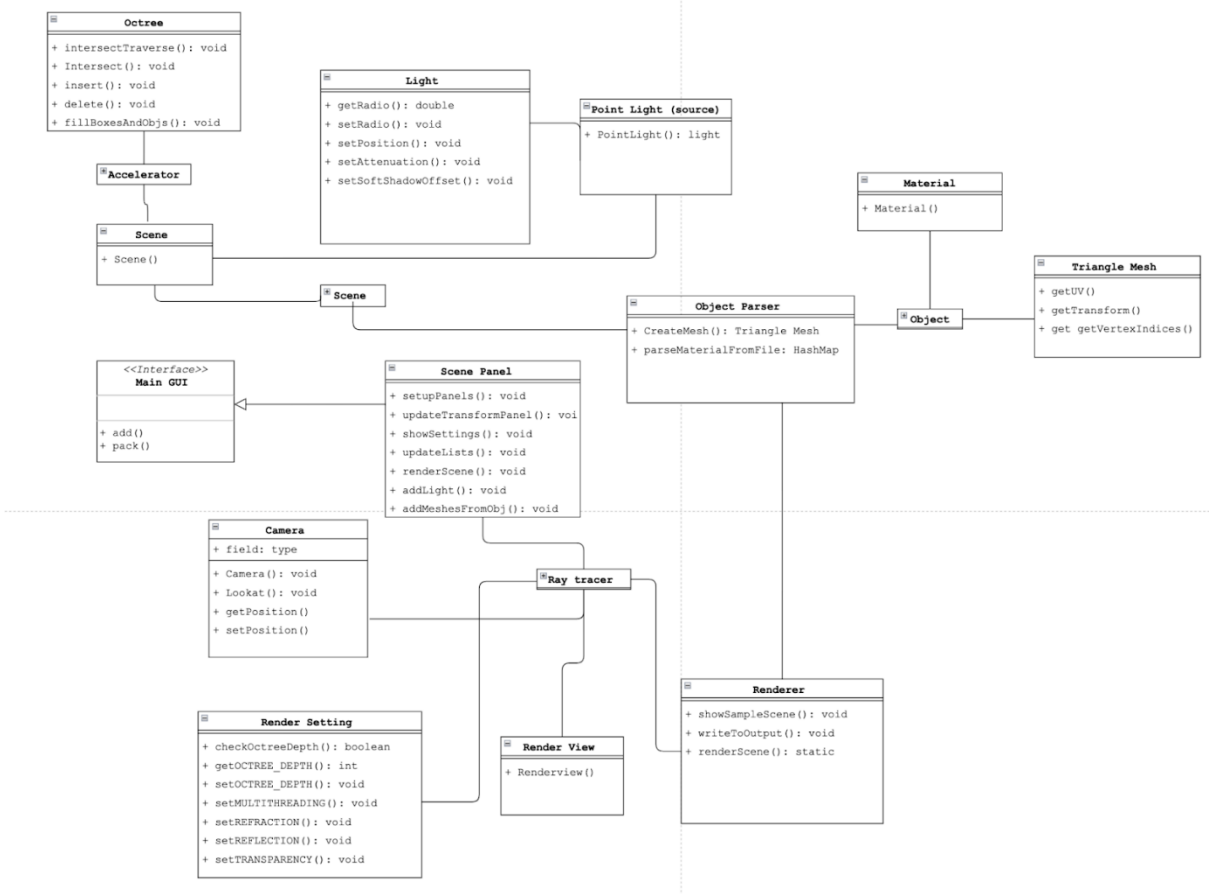

Figure 3.1: Class diagram of the project system

The UI of our project interface:

Main GUI:

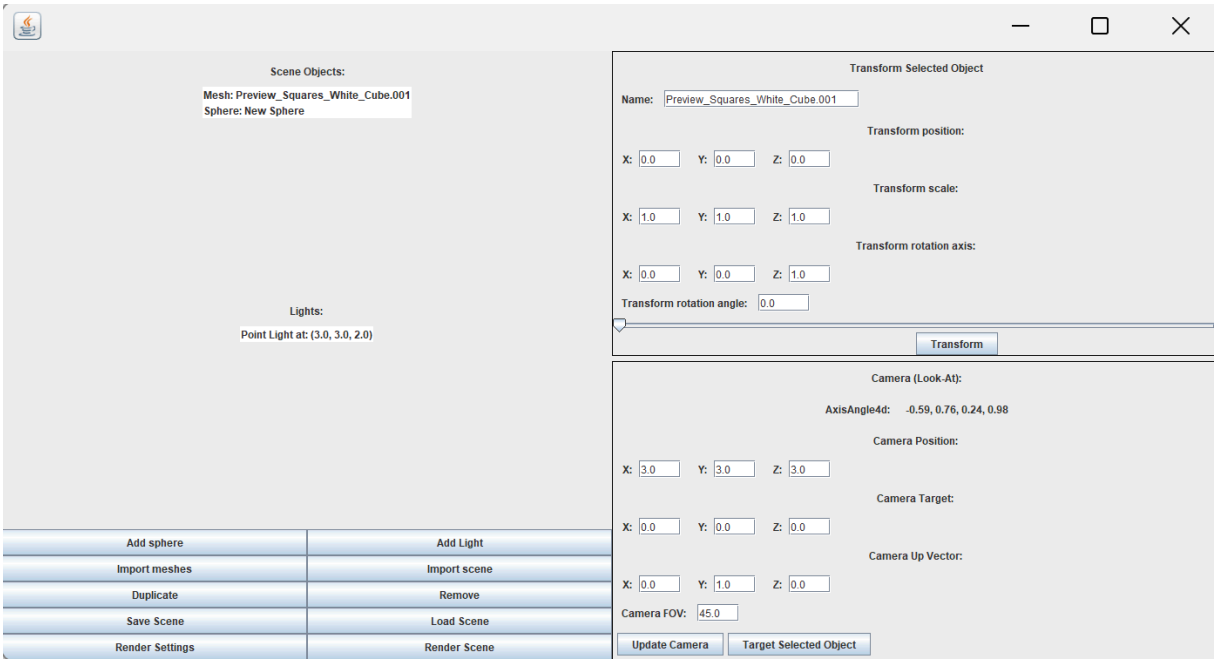

Figure 3.2: The user graphic interface of Render panel

Import meshes

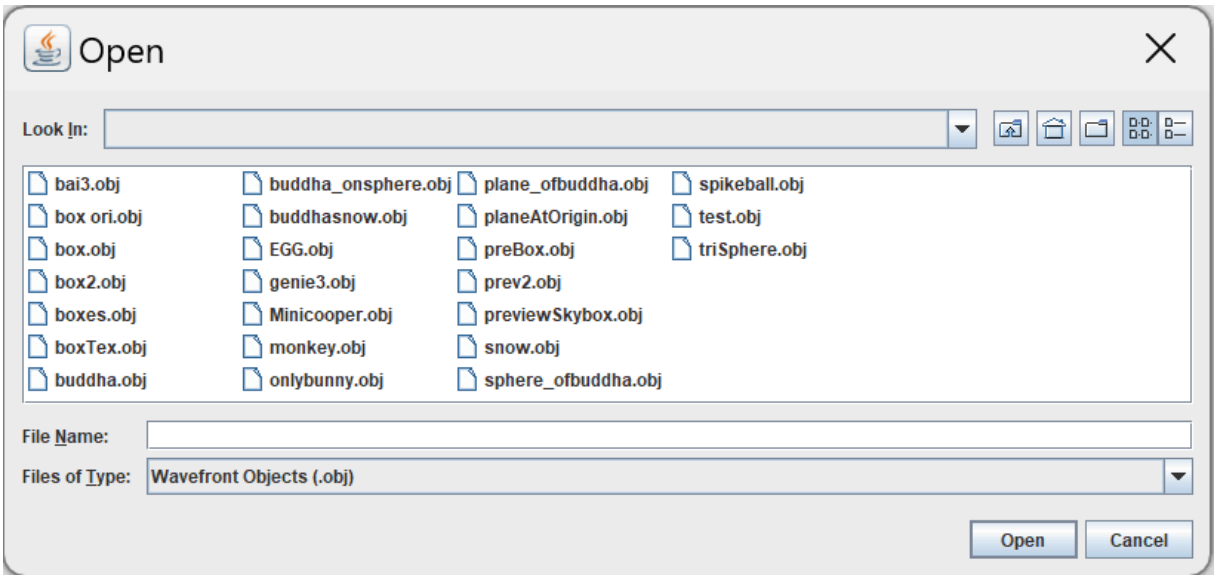

Figure 3.3: The user graphic interface of open object/scene panel

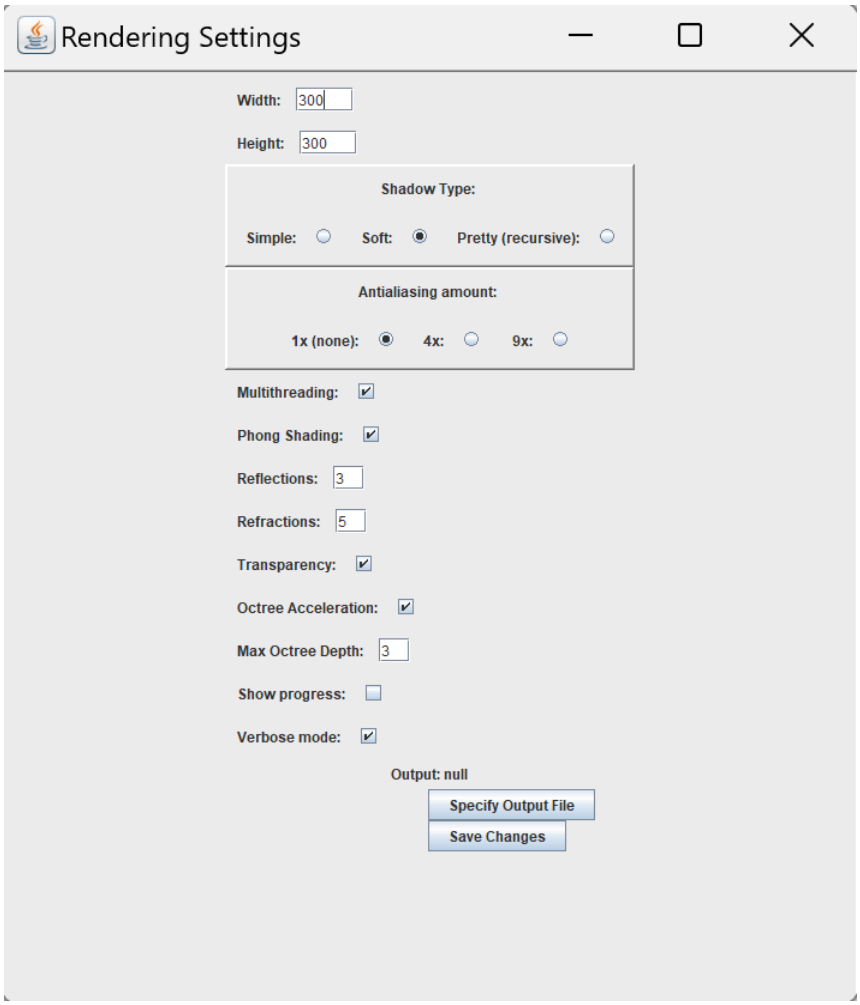

Figure 3.4: The user graphic interface of render setting panel

3.2.2. Diagram

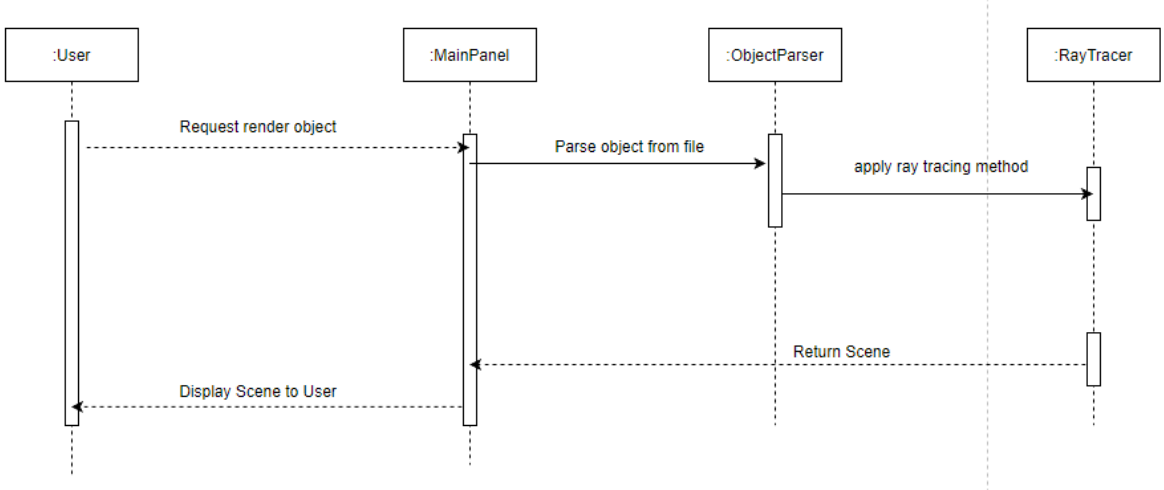

Figure 3.5: Sequence diagram of rendering procedure

CHAPTER 4

IMPLEMENT AND RESULTS

4.1. Phong Illumination

One of the critical parts of this project is how to calculate the color of points in the surfaces or objects to create and display the image, so we use Phong illumination model which presents the color of the object by three smaller components: Ambient, Diffusion and Specular light.

4.1.1. Ambient Light

Ambient light is a uniform component, It does not depend on the direction and surroundings around the scene. It means that any object which is hit by the primary rays will return its ambient light if there are light source lighten in it or not.Results

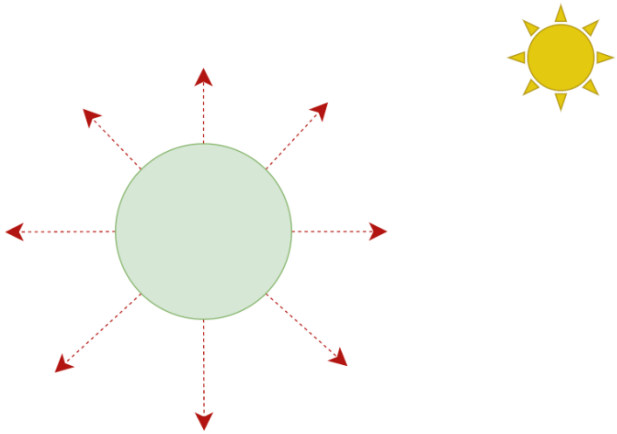

Figure 4.1: The example of Ambient light

4.1.2. Diffusion light

Diffusion light is the component of the object surface which is absorbed and then reflects the hit light ray in all directions, its intensity is not uniform and depends on the direction of the surface. It means if the light shines on the object, the object can return its diffusion component to the camera.

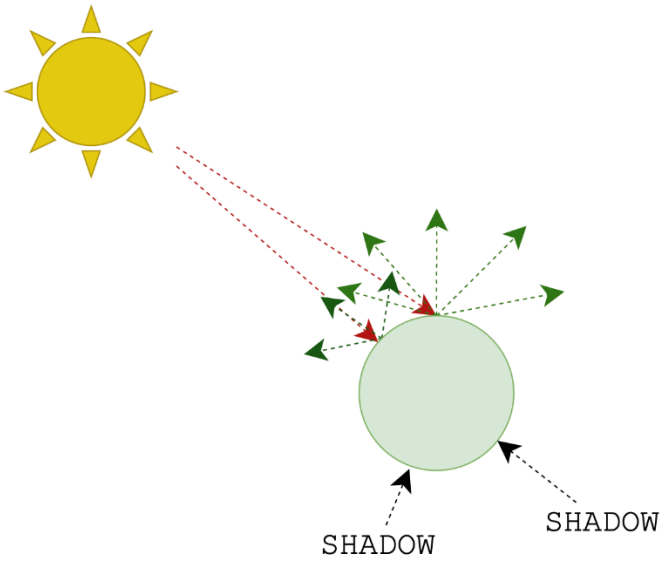

Figure 4.2: The example of Diffusion light

4.1.3. Specular light

Unlike Diffusion, Specular lighting does not reflect light in all directions. Instead, all light rays coming in the same direction will be reflected at the same angle. The diffusion component increases if the angle between the reflected light and the camera decreases.

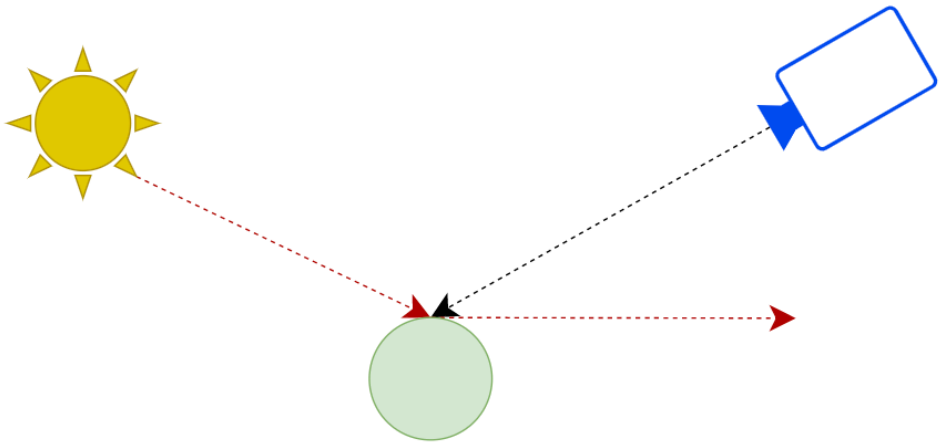

Figure 4.3: The example of Specular light

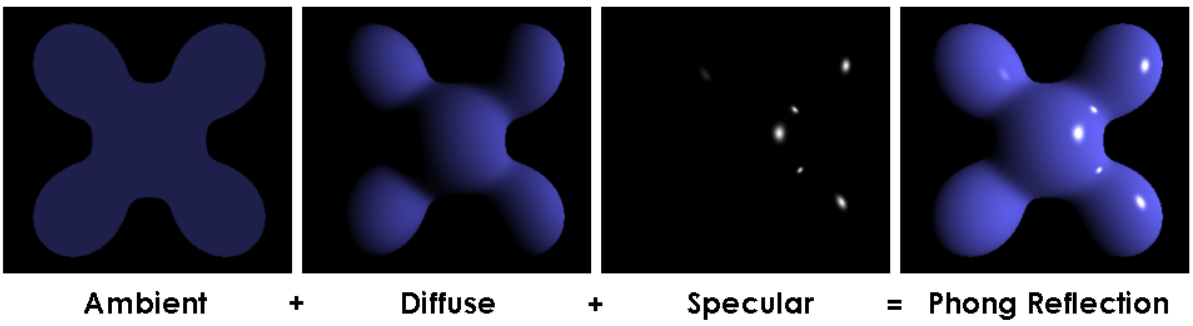

Figure 4.4: Combing 3 colour components = Phong Illumination

4.2. Secondary Rays

4.2.1. Reflection

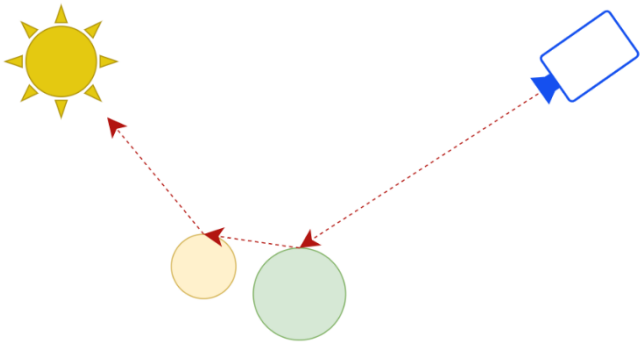

Figure 4.5: The example of reflection

Primary rays or camera's ray spawns a reflection ray whenever it hits a reflective object, these reflection rays then bounce off of the surface then collide with other objects, to solve the problems of reflection, the approach suggested by Bram de Greve of Stanford University is using Snell's law [11].

```
public static Ray reflectRay(Ray ray, Intersection i, double correction)
{
    Vector3d reflection = reflectVector(ray.direction, i.nn);
    Vector3d direction = new Vector3d(reflection);
    direction.scale(correction);
    Pt offsetPoint = new Pt(i.p);
    offsetPoint.add(direction);
    return new Ray(offsetPoint, new Vec(reflection));
}

public static Vector3d reflectVector(Vector3d incident, Vector3d normal)
{
        Vector3d in = new Vector3d(incident);
        Vector3d nor = new Vector3d(normal);
        //Snell's law
        double dotProduct = in.dot(nor);
        dotProduct *= 2;
        nor.scale(dotProduct);
        in.sub(nor);
        return in;
}
```

After getting the intersection point of the primary ray with the reflective object, the reflected ray is then calculated by its direction, offset by applying Snell's law.

4.2.2. Refraction

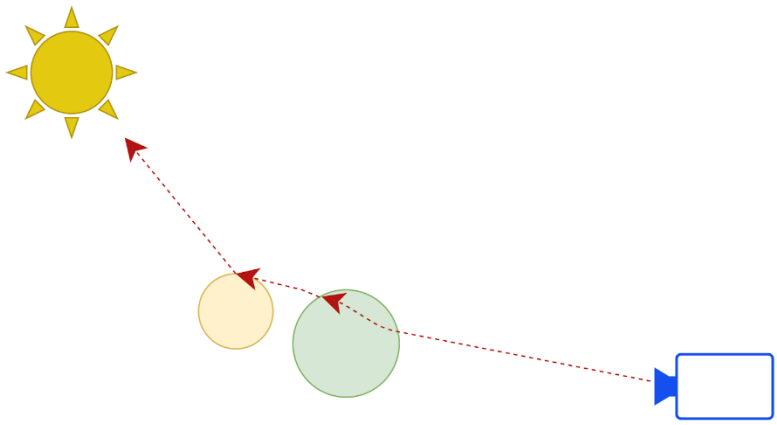

Figure 4.6: The example of refraction

Primary rays or camera ray spawns a refraction ray whenever they hit a transparent object, these refraction rays then move through the object with an angle determined by Snell's law, and when it hit the other side of the object, it then refracts again and collides with another object. Just as before, this problem can also be solved by applying Snell's law [11]

```
public static Ray refractRay(Ray ray, Intersection i, double
refractionIndex, double correction)
{
    Vector3d refraction = refractVector(ray.direction, i.nn,
refractionIndex, i.shape.getMaterial().refractionIndex);
    Vector3d direction = new Vector3d(refraction);
    direction.scale(correction);
    Pt offsetPoint = new Pt(i.p);
    offsetPoint.add(direction);
    return new Ray(offsetPoint, new Vec(refraction));
}
public static Vector3d refractVector(Vector3d incident, Vector3d normal,
double incidentIndex, double normalIndex){
    Vector3d inc = new Vector3d(incident);
    Vector3d norm = new Vector3d(normal);
    double n = incidentIndex / normalIndex;
    if (norm.dot(inc) > 0)
    {
        norm.negate();
    }
    double cosI = -norm.dot(inc);
    double sinT2 = n * n * (1.0 - cosI * cosI);
    if (sinT2 > 1.0)
    {
        System.out.print("Bad refraction vector!\n");
        System.exit(-1);
    }
    double cosT = Math.sqrt(1.0 - sinT2);
    inc.scale(n);
    norm.scale(n * cosI - cosT);
    inc.add(norm);
    return inc;
}
```

When the ray meets an object, the ray is then refracted at the point of intersection. The renderer then calculates the normal vector, the angle at which the vector and the ray are made along with the refraction angle, ray, ray's origin, ray's direction, and offset and then returns the refraction ray.

4.2.3. Recursive Shadow

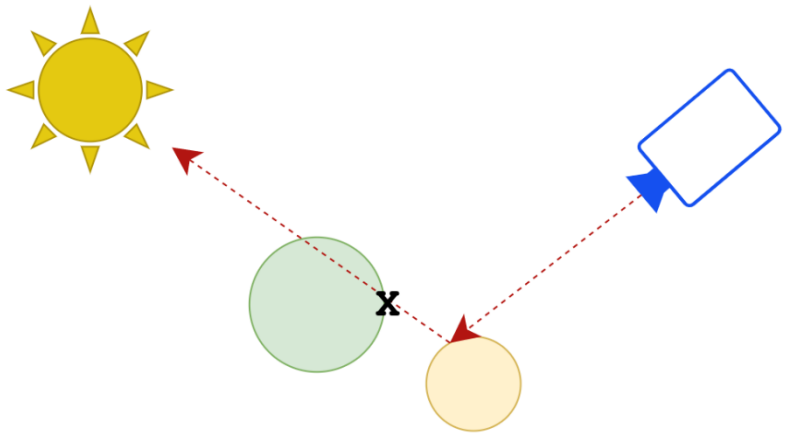

Figure 4.7: The example of recursive shadow

Another problem when attempting to recreate realistic lighting is the shadow of solid objects formed when light hits the object.

The solution is the same as the refraction problem, but in this case, instead of the light ray of a transparent object produces, shadow rays with the corresponding lighting are produced to simulate the lighting.

```
public static Ray makeShadowRay(Vector3d p, Light light, double
correction)
{
    Vec lightDir = new Vec(new Vector3d(light.getPosition()));
    lightDir.sub(p);
    Vector3d direction = new Vector3d(lightDir);
    direction.scale(correction);
    Pt offsetPoint = new Pt(p);
    offsetPoint.add(direction);

    return new Ray(offsetPoint, lightDir, 0);
}

public static ArrayList<Ray> makeShadowRays(Intersection inter, Light
```

```
light, double correction, double offset){
    ArrayList<Ray> shadows = new ArrayList<Ray>();
    ArrayList<Pt> points= new ArrayList<Pt>();
    Vector3d dU = new Vector3d(inter.dpdu);     dU.normalize();
    Vector3d dV = new Vector3d(inter.dpdu);     dV.normalize();
    Vector3d dUNeg = new Vector3d(dU);          dUNeg.normalize();
    Vector3d dVNeg = new Vector3d(dV);          dVNeg.normalize();
    dUNeg.negate();                             dVNeg.negate();

    Random r = new Random();
    r.setSeed(7);

    double randdv = r.nextDouble()*offset;
    double randdu = r.nextDouble()*offset;
    double randdvNeg = r.nextDouble()*offset;
    double randduNeg = r.nextDouble()*offset;
    dU.scale(randdu);
    dV.scale(randdv);
    dUNeg.scale(randdvNeg);
    dVNeg.scale(randduNeg);

    for (int i =0 ; i < 5; i++){
        points.add(new Pt(inter.p));
    }
    points.get(1).add(dU);
    points.get(2).add(dV);
    points.get(3).add(dUNeg);
    points.get(4).add(dVNeg);

    for (int i =0 ; i < points.size(); i++){
        shadows.add(makeShadowRay(points.get(i), light, correction));
    }

    return shadows;
}
```

This is the method that creates the shadow ray. It takes in the point of intersection, the light, and the correction factor. It then creates a vector from the light to the point of intersection. It then scales the vector by the correction factor and adds it to the point of intersection. This creates a new point that is offset from the point of intersection. It then creates a ray from the offset point to the light. [11]

4.2.4. Computation Time

With the refraction problem solved, there's also come to a different problem, which is how much would the refraction ray bounces

In the worst-case scenario, when every object is both transparent and reflective, all rays hit an object, it would lead to an O(infinity) case which would affect the rendering process to take a long time.

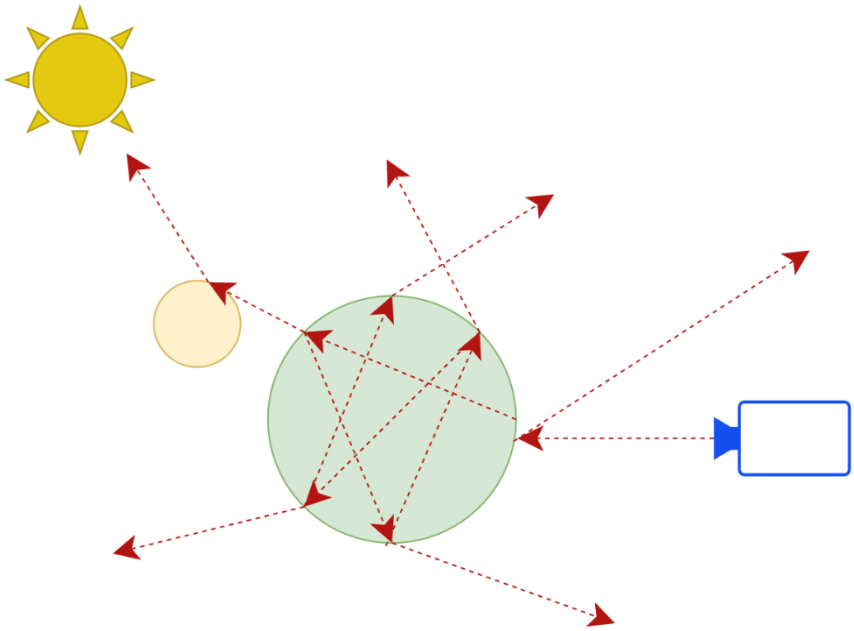

Figure 4.8: The example of refraction problem

```
public Ray(Ray r)
{
	this.position = new Pt(r.position);
	this.direction = new Vec(r.direction);
	maxt = r.maxt;
	mint = r.mint;
	depth = r.depth;
}
```

A simple approach to solve this problem is to limit the amount of time the refraction and reflection ray could bounce, which would greatly accelerate the computation time took before rendering the scene [12]

4.3. Optimization

4.3.1. Multithreading

As suggested by Graphica Computer Corporation, another approach to accelerate the rendering process is multithreading, which is using multiple threads to calculate the colour pixel. Since each pixel is calculated independently, it greatly decreases the computation time without adding any complexity in the system. [8]

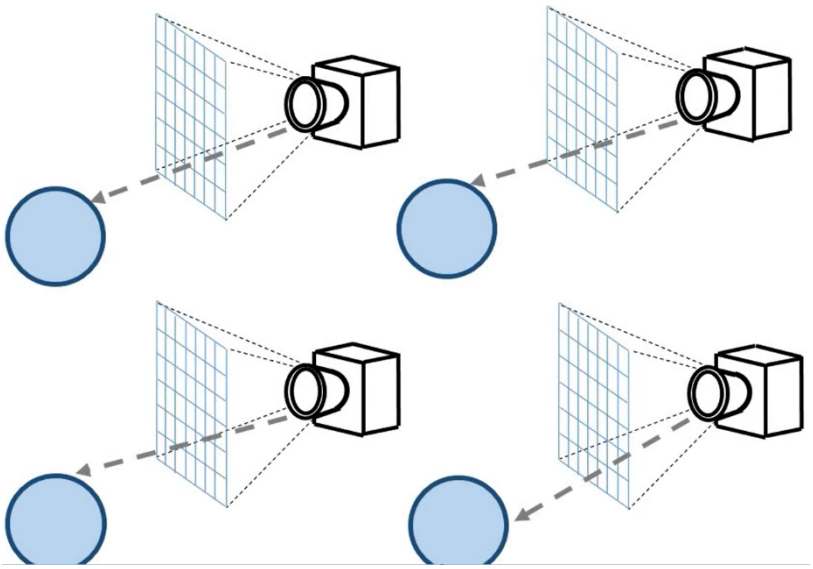

Figure 4.9: The example of multithreading

```
public BufferedImage renderThreads(Scene s) throws Exception
      {
            int height = s.settings.getHEIGHT();
            int width = s.settings.getWIDTH();
            double start = System.currentTimeMillis();
            rayCounter = 0;

            double startTime = System.currentTimeMillis();
            int totalRays = height * width;
            int currentRay = 0;

            final int NUM_THREADS;
            if (s.settings.isMULTITHREADING()) {
            NUM_THREADS = Runtime.getRuntime().availableProcessors() + 1;
            }
            else {
                  NUM_THREADS = 1;
            }
            if (s.settings.isVERBOSE()) {
                  System.out.println("Using " + NUM_THREADS + "
threads.");
            }

            final ExecutorService executor =
Executors.newFixedThreadPool(NUM_THREADS);
            Set<Future<ColorPoint>> futureSet = new
HashSet<Future<ColorPoint>>();
            Set<ColorPixel> queue = new HashSet<ColorPixel>();
            …
}
```

```
private class ColorPixel implements Callable
{
	int x;
	int y;
	Scene pixelScene;
	Octnode node = null;
	Ray ray = null;
	List<Ray> rays = null;
	Vector3d color = new Vector3d(0, 0, 0);
	double divisor = 1;
	RayTracer myTracer;
	…
}
```

In java, multithreading can be implemented using Callables, the primary ray casting is mapped by passing Callables to different threads then Callables return the specific colour pixel to render. This approach provides a linear performance increase with no effect on asymptotic complexity. However, this approach's computation time is dependent on the number of CPU cores available.

4.3.2. Octree

Another approach suggested by Whang et al to accelerate the computation, is using a data structure which is Octree. Here, instead of the camera trying to shoot out as many rays as possible to hit an object, the scene is divided into quadrants called octnodes. Scene objects are then placed into the octnodes to accelerate intersection operation. When an object is placed into an octnodes, it can be split into eight child octnodes, which may or may not contain the object. [9]

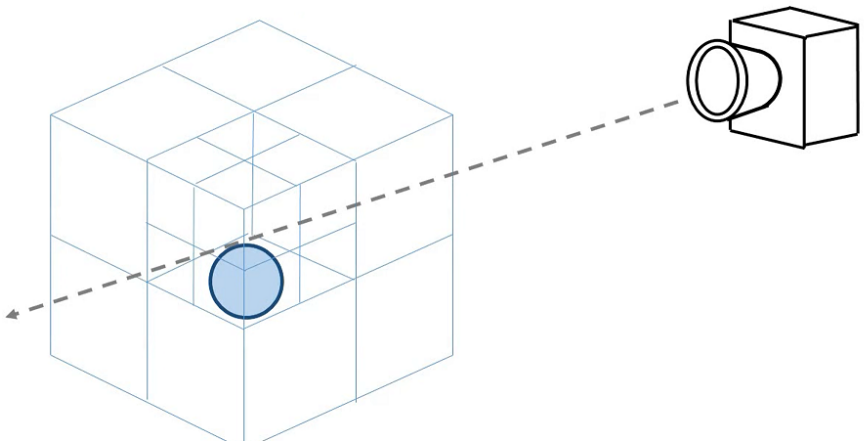

Figure 4.10: The example of octree traversal

The renderer would check the ray for collision with the octree, then recursively check its children in each layer. Octnodes are only checked if they contain objects.

```
      public Octree(Scene scn, int maxdepth) throws RefinementException,
SplitBeyondMaxDepthException
      {
            BBox rootBox = new BBox();
            ArrayList<BBox> scnBoxes = new ArrayList<BBox>();
            ArrayList<SceneObject> objs = new ArrayList<SceneObject>();
            ArrayList<SceneObject> tmpObjs = new
ArrayList<SceneObject>();

            for (SceneObject obj : scn.getObjects())
            {
                  fillBoxesAndObjs(obj, objs, tmpObjs, scnBoxes);
                  rootBox = BBox.union(rootBox, obj.getWorldBound());
            }
            rootBox.expand(scnBoxEpsilon);
            root = new Octnode(rootBox, 0, maxdepth);
            for (int i = 0; i < objs.size(); i++)
            {
                  root.insert(objs.get(i), scnBoxes.get(i));
            }
      }
```

4.4. Results

This section will show the result of our rendering system on different scenes, include:

Reflective sphere, Stanford's Buddha model, Spheres in room, Stanford's Bunny model

4.4.1. Reflective sphere

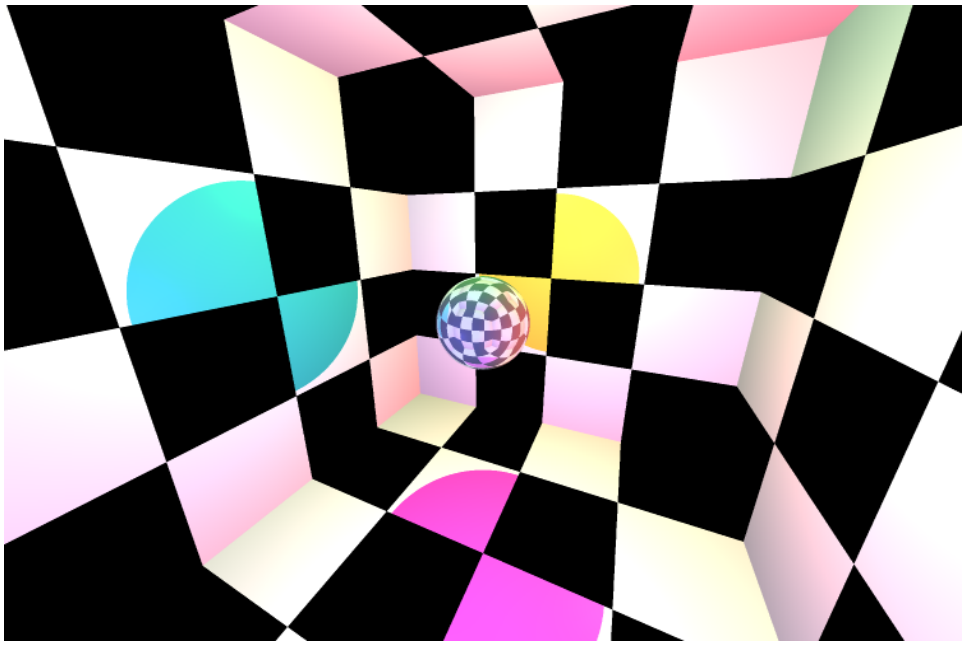

Figure 4.11: Reflective sphere with 3 light sources

Primitive	No acceleration	Multithreading	Octree	Multithreading and Octree
375000 rays	12.652 seconds	4.901 seconds	10.575 seconds	3.817 seconds

Table 4.1: Reflective sphere's result

4.4.2. Stanford's Buddha model

Figure 4.12: Stanford's Buddha model

Primitive	No acceleration	Multithreading	Octree	Multithreading and Octree
375000 rays	400.972 seconds	215.422 seconds	4.508 seconds	1.621 seconds

Table 4.2: Stanford's Buddha model's result

4.4.3. Spheres in room

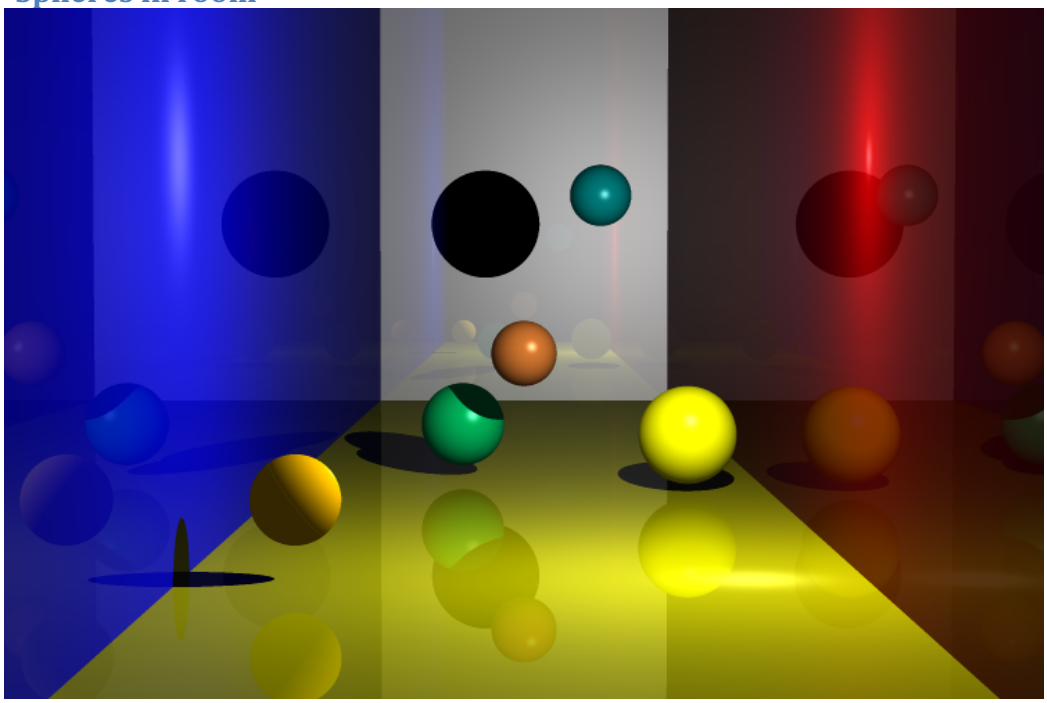

Figure 4.13: Spheres in room

Primitive	No acceleration	Multithreading	Octree	Multithreading and Octree
375000 rays	5.539 seconds	2.963 seconds	16.604 seconds	5.786 seconds

Table 4.3: Spheres in room's result

4.4.4. Stanford's Bunny model

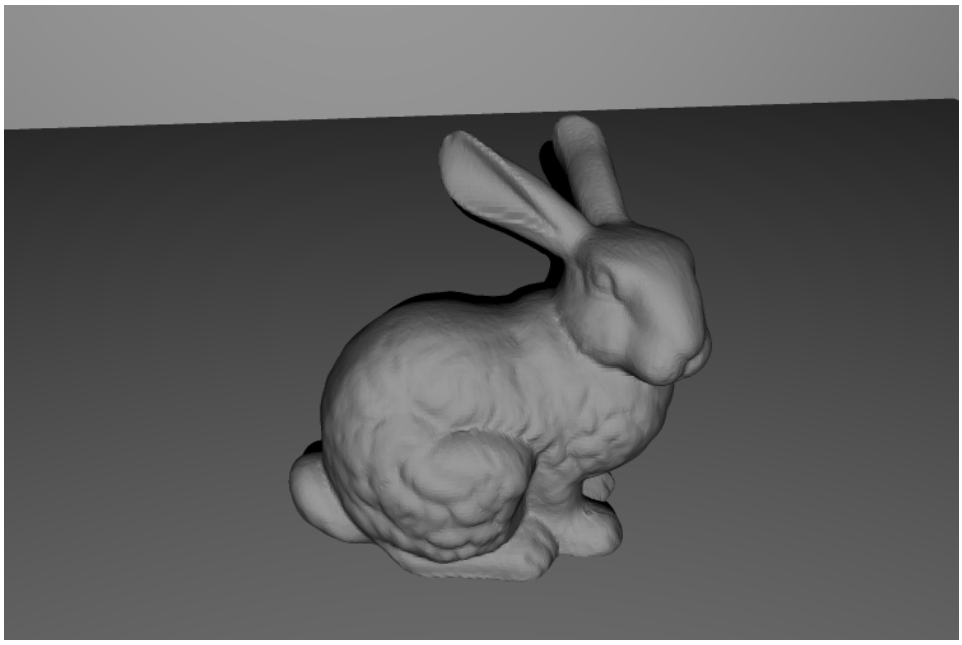

Figure 4.14: Stanford's Bunny model

Primitive	No acceleration	Multithreading	Octree	Multithreading and Octree
375000 rays	4363.337 seconds	1654.255 seconds	1236.865 seconds	337.645 seconds

Table 4.4: Stanford's Bunny model's result

CHAPTER 5

DISCUSSION AND COMPARISON

5.1. Discussion

	Primitive	No acceleration	Multithreading	Octree	Multithreading and Octree
Reflective sphere	375000 rays	12.652 seconds	4.901 seconds	10.575 seconds	3.817 seconds
Buddha	375000 rays	400.972 seconds	215.422 seconds	4.508 seconds	1.621 seconds
Spheres in room	375000 rays	5.539 seconds	2.963 seconds	16.604 seconds	5.786 seconds
Bunny	375000 rays	4363.337 seconds	1654.255 seconds	1236.865 seconds	337.645 seconds

Table 5.1: Result comparison between scenes

Our result indicates that the system can perform rendering polygon scenes and optimise the computational time with different methods such as multithreading on the hardware support, octree in the data structure and these two combined. After analysing multiple scenarios, the results have shown that with no software acceleration process, it would generally take longer for a scene to be rendered. However, that's not always the case; for example, in the case of Figure 4.13, applying octree took a longer time to render out compared to other methods. In all cases, the results indicate that the most efficient method of rendering is to use both Multithreading and Octre, with it having a much lower computation time. The results fit with the theory that when applying multithreading and octree approaches, we can speed up the ray tracing and reduce the computation time for rendering a specific object. The results give us a drawback when the object becomes more complex, the computational time takes too long to finish; for example, the bunny object is a simple object, but in our system, it can be considered a complex rendered scene due to its long time computation.

Further research is needed to focus on reducing the computation time for rendering the objects as well as establishing photo-realism rendering. Therefore, we can obtain a more realistic renderer, create more sophisticated objects and improve our system performance.

5.2. Comparison

In this section, we try to compare two popular renderer models to create realistic images, that is, ray tracing and path tracing. While ray tracing works by emitting rays from the camera through each pixel of the virtual screen and traces the path of those rays in the scene by algorithm, path tracing works by emitting numerous rays through each pixel at the same time so we can receive the better quality of the scene. In the game industry, ray tracing is more favoured because it can create more realistic images, but the cost for computation is higher. When working with a more sophisticated scene which contains many objects in it, it is found that path tracing is more efficient than ray tracing. Although ray tracing gives us a realistic image, it needs a lot of power from the CPU and the GPU, so the cost to obtain ray tracing is prohibitive.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1. Conclusion

Our raytracing technique that has been implemented above shows us how the render systems work and also lets us understand the basic principle of how to construct an object in the environment and how the light, especially the Phong illumination, affects the visualisation of a scene. In this work, we also implement some optimisation of a render system. While quicker raster-based shading techniques have been favoured for video games to satisfy real-time requirements, ray tracing has long been employed in motion pictures to create photorealistic graphics. Recent Graphics Processing Units (GPUs) do, however, include hardware accelerators for ray tracing. The traversal of hierarchical tree data structures, which is utilised to check for ray-object crossings, is the focus of these accelerator units. Numerous redundant ray-box intersection checks are carried out by rays travelling down the same routes through these constructions. [14] The result of our system remains inefficient when applied to real-time rendering systems that may appear in some applications: Blender, Maya or video games,... but, as a final word, we hope that our work can be an informative reference that can help others to understand a rendering procedure from scratch.

6.2. Future work

In the future, we will optimise the computation time of the rendering process by trying another optimized algorithm, and we also will try to apply more new efficient methods to deal with complex scenes and long computational time.

REFERENCES

[1]. Wald, I., Mark, W. R., Günther, J., Boulos, S., Ize, T., Hunt, W., ... & Shirley, P. (2009, September). State of the art in ray tracing animated scenes. In Computer graphics forum (Vol. 28, No. 6, pp. 1691-1722). Oxford, UK: Blackwell Publishing Ltd.

[2]. Wald, I., Slusallek, P., Benthin, C., & Wagner, M. (2001, September). Interactive rendering with coherent ray tracing. In Computer graphics forum (Vol. 20, No. 3, pp. 153-165). Oxford, UK and Boston, USA: Blackwell Publishers Ltd.

[3]. Liang, X., Yang, H., Qian, Y., & Zhang, Y. (2014). A Fast Kd-tree Construction for Ray Tracing based on Efficient Ray Distribution. J. Softw., 9(3), 596-604.

[4]. Rüger, A. (1993). Dynamic ray tracing and its application in triangulated media. 1990-1999-Mines Theses & Dissertations.

[5]. STOLL G.: Part II: Achieving real time/optimization techniques. Slides from the Siggraph 2005 Course on Interactive Ray Tracing, 2005.

[6]. Rüger, A. (1993). Dynamic ray tracing and its application in triangulated media. 1990-1999-Mines Theses & Dissertations.(2.2.2)

[7]. Christen, M. (2005). Ray tracing on GPU. Master's thesis, Univ. of Applied Sciences Basel (FHBB), Jan, 19.

[8]. Fujimoto, A., Tanaka, T., & Iwata, K. (1986). Arts: Accelerated ray-tracing system. IEEE Computer Graphics and Applications, 6(4).

[9]. Whang, K. Y., Song, J. W., Chang, J. W., Kim, J. Y., Cho, W. S., Park, C. M., & Song, I. Y. (1995). Octree-R: An adaptive octree for efficient ray tracing. IEEE Transactions on Visualization and Computer Graphics, 1(4).

[10]. Friedrich, H., Günther, J., Dietrich, A., Scherbaum, M., Seidel, H. P., & Slusallek, P. (2006, July). Exploring the use of ray tracing for future games. In Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames.

[11]. De Greve, B. (2006). Reflections and refractions in ray tracing. Retrived Oct, 16, 2014.

[12]. Weghorst, H., Hooper, G., & Greenberg, D. P. (1984). Improved computational methods for ray tracing. ACM Transactions on Graphics (TOG), 3(1), 52-69.

[13]. Wikimedia Foundation. (2022, April 9). Phong reflection model. Wikipedia. Retrieved November 28, 2022, from https://en.wikipedia.org/wiki/Phong_reflection_model

[14]. Liu, L., Chang, W., Demoullin, F., Chou, Y. H., Saed, M., Pankratz, D., ... & Aamodt, T. M. (2021, October). Intersection Prediction for Accelerated GPU Ray Tracing. In MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (pp. 709-723).

[15]. Staff, H. C. (2022, October 13). Ray Tracing vs. path-tracing: What's the difference? History. Retrieved November 28, 2022, from https://history-computer.com/ray-tracing-vs-path-tracing-whats-the-difference/

[16]. Djeu, P., Hunt, W., Wang, R., Elhassan, I., Stoll, G., & Mark, W. R. (2011). Razor: An architecture for dynamic multiresolution ray tracing. ACM Transactions on Graphics (TOG), 30(5).

[17]. Benthin, C., Wald, I., & Slusallek, P. (2004, November). Interactive ray tracing of free-form surfaces. In Proceedings of the 3rd international conference on Computer graphics, virtual reality, visualisation and interaction in Africa.

[18]. Stephens, A., Boulos, S., Bigler, J., Wald, I., & Parker, S. G. (2006, May). An Application of Scalable Massive Model Interaction using Shared-Memory Systems. In EGPGV@ EuroVis/EGVE (pp. 19-26).

[19]. Havran, K. D. V., & Seidel, H. P. Faster Ray Tracing with SIMD Shaft Culling.

[20]. A. Reshetov. Faster Ray Packets-Triangle Intersection through Vertex Culling. Proceedings of the IEEE Symposium on Interactive Ray Tracing, pages 105-112, 2007.
[21]. Goldsmith, J., & Salmon, J. (1987). Automatic creation of object hierarchies for ray tracing. IEEE Computer Graphics and Applications, 7(5), 14-20.
[22]. Müller, G., & Fellner, D. W. (1999, February). Hybrid scene structuring with application to ray tracing. In Proceedings of the International Conference on Visual Computing (ICVC'99) (pp. 19-26).

APPENDIX

Antialiasing

While implementing the multithreading accelerator as suggested in the science paper [8], it also talked about the method of improving the quality of the rendered image using a technique called Super Sampling Anti Aliasing (SSAA). This method shoots multiple rays per pixel to help smooth the image overall.

Again, by applying it to our system, the image quality improved but with a cost to the render time since it would shoot out multiple rays compared to the primitive ones.

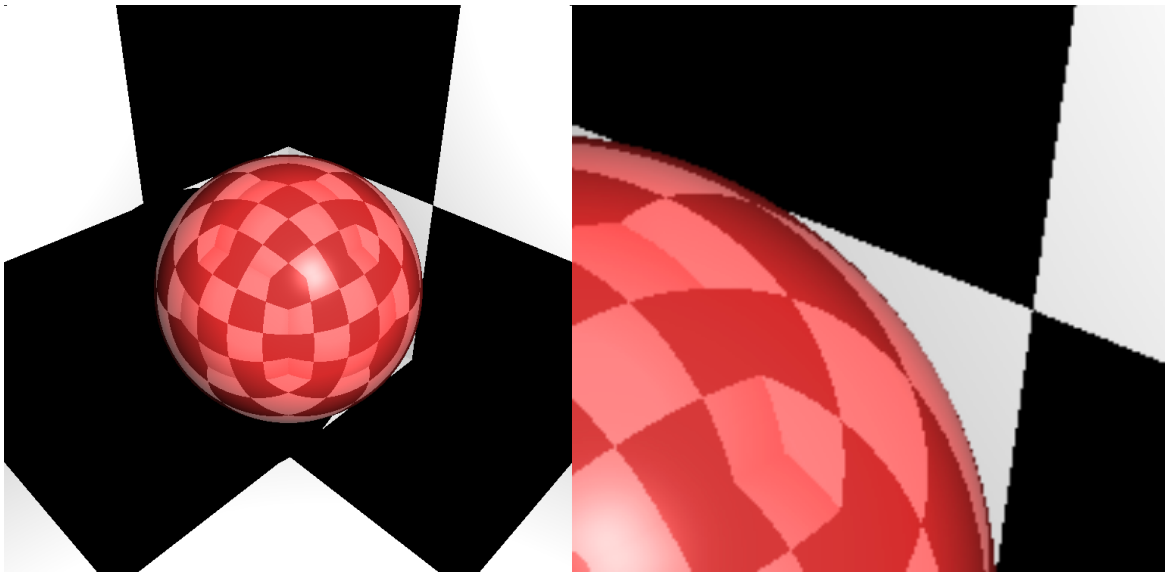

Figure 7.1: Reflective sphere with no antialiasing

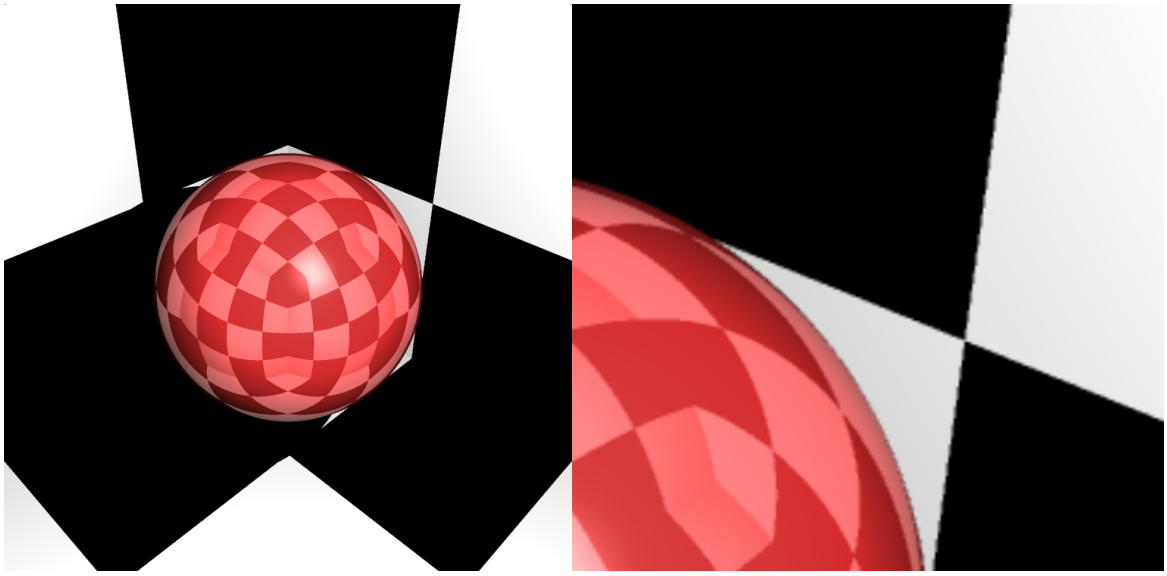

Figure 7.2: Reflective sphere with x4 antialiasing

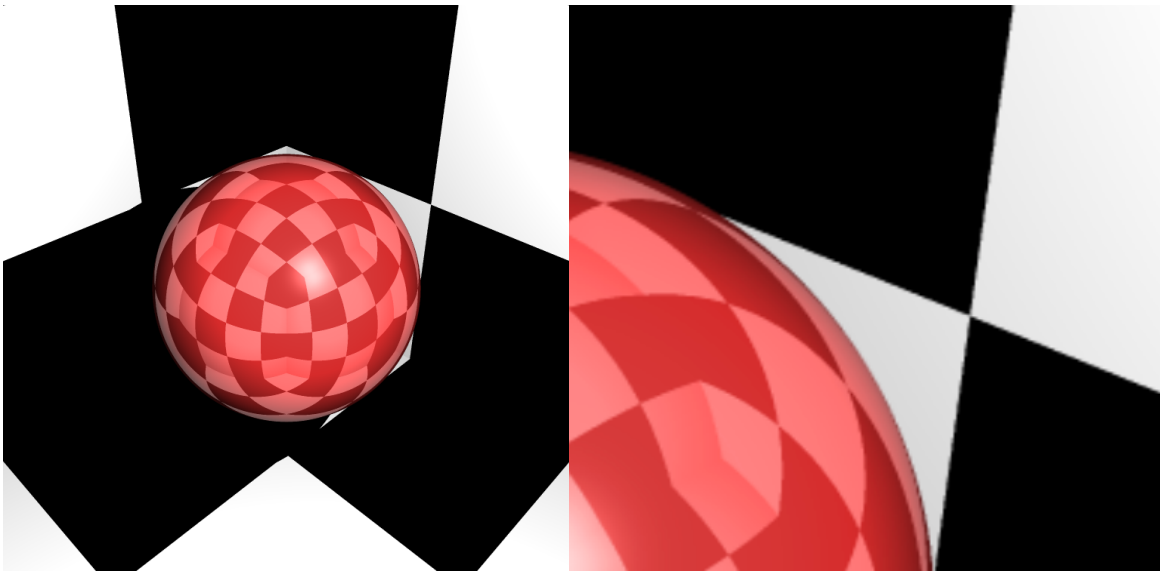

Figure 7.3: Reflective sphere with x9 antialiasing