

Master Project

Computer Graphics: Rendering Track

Alhajras Algdairy

Advisor: Prof. Dr.-Ing. Matthias Teschner

September 23, 2021

Acknowledgment

I want to express my special thanks of gratitude to Prof. Dr.-Ing. Matthias Teschner, who guided me through my master's program in general and in this master's project in specific. This project gave me the golden opportunity to dig into an exciting topic in computer graphics that I am keen on. The flexibility in research made me learn more about the related topics I am interested in, in rendering. Reading scientific topics and reviewing codes from different repositories expanded my expertise, where I had a full responsibility to control my time and resources, to learn more about the research process.

Abstract

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

1 Introduction

This report summarises my journey to implement a simple raytracer focusing on data structure level, where the performance of rendering a scene will be the lion's share of the report. Theory, implementation, and results will be discussed in depth in the report, where an introduction and motivation of what is raytracer and how to implement it will be briefly discussed; because data structures are more interesting for us, I will only explain the topics that are used in this raytracer.

2 What is Raytrcrer and how it works

Computer graphics has three main pillars: *Modelling*, *Rendering*, and *Simulation*. Scientists are interested in simulating a real-life phenomenon, such as Gravity Fraction, Rain, Snow, and the exciting part for us, light. Simulating light is arguably the most challenging part because light has always been difficult to characterize as it can behave as particles and waves, which

makes it spread into the whole scene based on probabilities. This makes it difficult to compute as it involves complex computation and infinite simulations to execute to have a perfect result without an error, this is what is known as Rendering.

Why do we need to simulate light? Adding light into a scene will generate shadows, reflection, and refraction consequently will illuminate the scene, making the scene look like a reality, which can be helpful for some applications. For the internal designers, it is vital to simulate the final result of the lightning inside a room, such as sunlight coming from the windows, the light of lamps, and fireplace, these with react together and create shadows, before constructing the building and payloads of money, simulating the right angle and place of the room is helpful to imagine the final result. Solar engineers use 3D tools to build solar panel farms where the angle between the sunlight and the panel is essential to gather as much light energy as possible. In gaming, different companies compete to design engines that can produce natural scenes; this includes lighting and shadowing.

Figure 5, shows an example of rending a scene that can not be distinguished from reality, the details it catches as glossy materials, the reflection of surfaces, and the shadow. Capturing these details requires rendering techniques. Two popular methods are Rasterization and Raytracing. Raytracing outperforms rasterization in capturing more details; however this leads to performance issues, that is why most application uses Raytracing is offline rendering and not real-time rendering like games, where speed is vital to render each frame with compromising details.



Figure 1: <https://www.pcgamer.com/unreal-engine-5-tech-demo-pc-performance/> The raytracing algorithm builds an image by extending rays into a scene and bouncing them off surfaces and towards sources of light to approximate the color value of pixels [Piotr Dubla, "Interactive Global Illumination on the CPU."]

2.1 Raytracing definition

So what is Raytracing? Raytracing is a rendering technique that provides highly lifelike lighting effects. In other words, an algorithm can track the source of light and then mimic how

the light interacts with the virtual objects it eventually encounters in the computer-generated environment. Raytracing produces far more lifelike shadows and reflections, as well as significantly enhanced translucence and dispersion. The algorithm considers where the light falls and calculates the interaction and interplay in the same way as the human eye does with actual light, shadows, and reflections.

2.2 Raytracing mechanism

Raytracing follows three main steps: *Casting Rays*: This is similar to how the eye works, however rather than eyes works as a sensor, we have a camera, we shoot rays from the camera to the scene, and we calculate which is the closest object it hits, its color and brightness. This is done for each pixel. *Path tracing*: Casting rays can solve the visibility issue; this means which object appears on the camera and at what position and color. However, if we want to simulate the following effects: soft shadows, depth of field, motion blur, caustics, ambient occlusion, and indirect lighting, then we need to trace the rays from or to the light source, this will accumulate the brightness of an object or model in a scene, also will give the depth of different objects. Note that the more rays and depth we need, the more quality we get but the slow the simulation becomes due to the complexity. *Shading*: In addition to tracing rays, we need a model to simulate different materials like Transparent, Glossy, and Diffuse. This is where the Shading phase is needed.

Raytracing has two main methods:

- *Forward Raytracing*: The light particles (photons) are tracked from the light source to the object via Forward Raytracing. While forward ray tracing is the most exact method for determining the color of each object, it is also the most inefficient.
- *Backward Raytracing*: An eye ray is formed at the eye in backward ray tracing, and it travels through the viewplane and out into the scene. If it hits an object, it will return it to the viewplane immediately. This method is more efficient than Forward Raytracing but less accurate due to reducing the rays used. In this implementation this method is used.

3 Raytracer Implementation

In this chapter, raytracer terms and definitions are introduced, illustrating the implementation and decisions that have been made. We need to set up our implementation to work for the raytracer as shown in Figure 2; the basic raytracer requires a camera; its responsibility is to shoot rays that travel through the scene and return a value or color. How many rays should it shoot? This depends on the width and height of the image we want to generate, the bigger the image, the more details we capture, but the more expensive computation gets. The rays idea is to find an intersection with the scene objects and try to return a color of the object to represent a corresponding pixel value. For object visibilities, we do not need a light source; however, to make the scene more realistic, we need to add shadows and other effects as reflections and refractions; hence we need a source light and tools, as shown in Figure 2.

The algorithm works as follows, initiating the camera position and the orientation direction or where it looks at. Secondly, we subdivide the scene into pixels; this is based on the settings of the raytracer, usually by setting a height and width, reading a scene from usually an XML file that has a description of the number of objects in the scene, their color, and type. Afterward, generating rays for each pixel, this ray will be shoot toward the scene, and for each ray, we go

through all the objects and test intersection tests; if the test returns true, we save its position, and we save the object, we do this for all objects and if more than two objects are intersected we compare between their position to detect which one is closer to the camera. In order to create shadows, we track the intersection point from the camera toward the light; if there is an object between the light source the hitting point, then we detect shadow. The corresponding value of the pixel is saved in a buffer and after the ray tracing is done, the pixels value are saved in a file of PPM extension.

This is the basic idea, and to improve it, other topics need to be covered, such as Data structures, Objects Materials, Anti aliasing, Soft shadow. In this chapter, the basic blocks and components that make up the raytracer will be explained; only the methods are used will be explained, because as mentioned before, the performance of the raytracer is the main scope for this project.

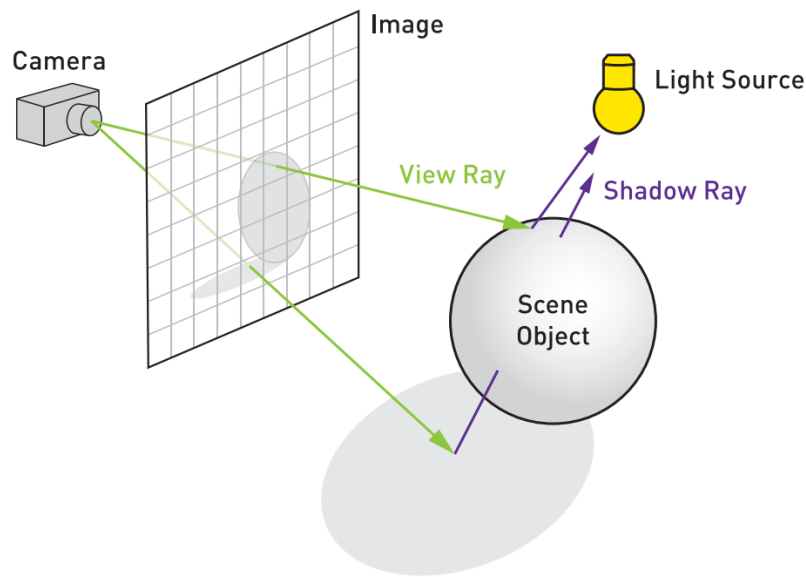


Figure 2: An image showing how a ray is casted from the camera and how objects are illuminated and shadows are computed in ray tracing .

3.1 Software architecture

Configuration

We start with the configuration of the Raytracer by introducing a **struct Settings**, here we specify the following properties of the raytracer such as the **width** of the image, **height** of the image, the **background default color** of the scene, **maximum depth of path tracing**, **anti-aliasing samples** and **acceleration data Structure type**. Some of the previous terms will be explained later.

Scene generation

To create a scene, spheres will be used to create meshes; mesh is a collection of objects at a position and orientation to build a much more complex object. For example, the Stanford bunny shown in Figure 3 is composed of approximately 35947 spheres. Usually, meshes are made of triangles because they are more efficient and easier to build a mesh; however, I want

to try to use something else and test the result and the quality of spheres. `createScene()` is implemented to load .obj files the next models will be used for testing in this report: **Stanford Bunny**, **Igea** and **Armadillo**, Figure 4 shows the different complex model that will be used for testing.

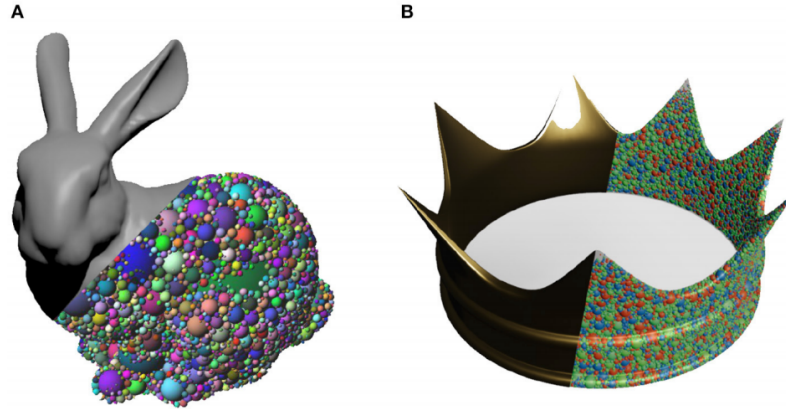


Figure 3: Stanford bunny and a crown made of a collection of spheres .

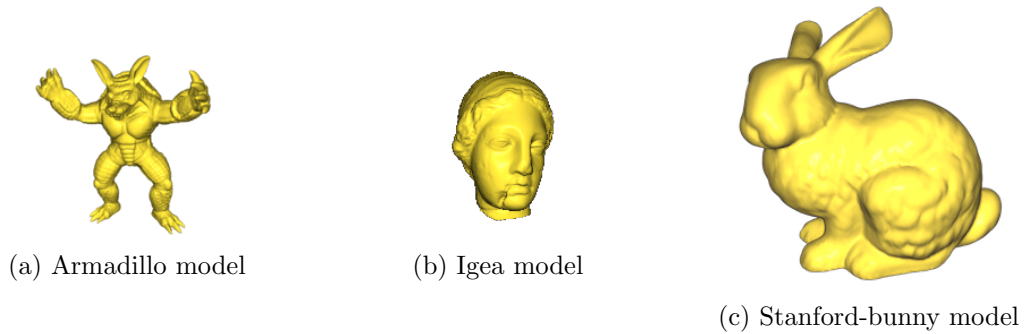


Figure 4: Rendering different shapes in the scene

The **Sphere** is a class with the next attributes: **Center**: Position of the sphere, **MaterialType**: *diffuse*, *glossy*, *reflection* and *refraction*, **Radius**: Sphere radius usually 1, **surfaceColor**: Color of the sphere in **RGB** format, and **emissionColor**: This is for the light. Lights are just vector of spheres where the **emissionColor** is set to one. The light emission is using the **inverse-square law**. In science, an inverse-square law is any scientific law stating that a specified physical quantity is inversely proportional to the square of the distance from the source of that physical quantity. In other words, the more significant the distance between the light source and the object, the less light it reaches the object surface.

For point lights, this can be formulated as:

$$l_{ip} = \frac{l_i}{4 * \pi * d^2} \quad (1)$$

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

Shooting rays

For this, all raytracers use rays as a way to simulate photons. Let us think of a ray as a function; here \mathbf{p} is a 3D position along a line in 3D. \mathbf{o} is the ray origin and \mathbf{d} is the ray direction.

$$\mathbf{p}(t) = \mathbf{o} + t\mathbf{d}, \quad 0 < t < \infty \quad (2)$$

Figure 5 shows how Rays are used and how equations 1 can be used to check intersection tests and calculate the distance between the origin ray point, the camera, and the intersection point. Point distance d is used to compare the points to check which object is closer to the camera.

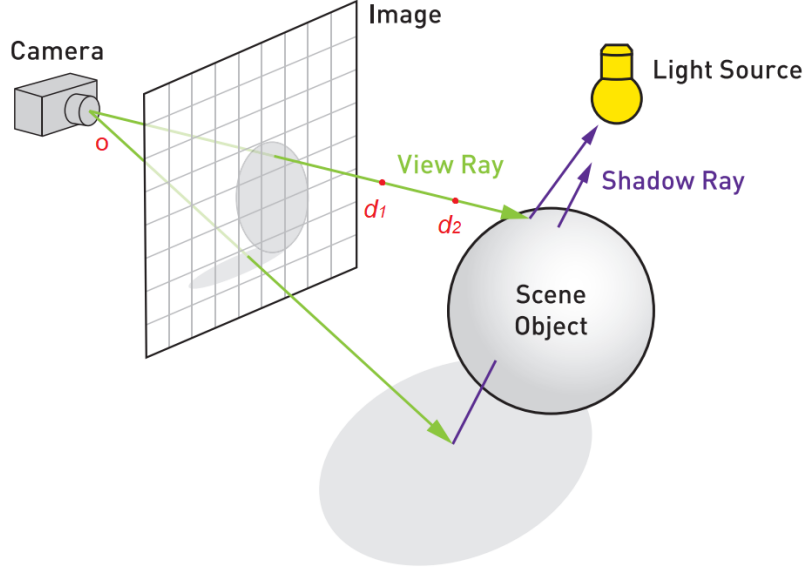


Figure 5: An image showing how a ray is casted from the camera and how objects are illuminated and shadows are computed in ray tracing .

Rendering Sphere

For testing, spheres are often used in ray tracers because calculating whether a ray hits a sphere is pretty straightforward.

The general equation of a sphere with radius = 1 is:

$$(o_x + td_x)^2 + (o_y + td_y)^2 + (o_z + td_z)^2 = 1 \quad (3)$$

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

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

By solving the equation we get three different cases as shown in Figure 6:

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

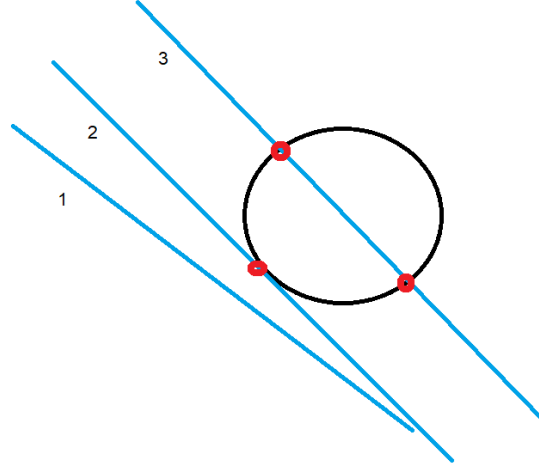


Figure 6: The three possible line-sphere intersections: 1. No intersection. 2. Single point intersection. 3. Two point intersection.

Shading

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

The computed light at a point P depends on the following:

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

3.1.1 Lambert's Cosine Law

The amount of light energy arriving at a surface is proportional to the cosine of the angle between the light direction and the surface normal, according to *Lambert's cosine law*. Illumination strength at a surface is proportional to the cosine of the angle between \mathbf{l} and \mathbf{n} , the angel will

be denoted as θ , the following three cases illustrate the relationship between the \mathbf{L}^{source} and $\mathbf{L}^{surface}$.

The $\mathbf{L}^{surface}, \mathbf{L}^{source}$ relation is:

$$\mathbf{L}^{surface} = \mathbf{L}^{source} \cdot \cos \theta \quad (5)$$

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

3.1.2 Phong reflection model

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

• Ambient reflection

$$\mathbf{L}^{amb} = \boldsymbol{\rho} \otimes \mathbf{L}^{indirect} \quad (6)$$

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

Diffuse reflection

$$\mathbf{L}^{diff} = \mathbf{L}^{source} \cdot (\mathbf{n} \cdot \mathbf{l}) \otimes \boldsymbol{\rho} \quad (7)$$

- \mathbf{L}^{source} , is the light source color and intensity which usually white.
- \mathbf{n} and \mathbf{l} , are the representation of the Lambert's cosine law, where \mathbf{n} is the normal surface vector and \mathbf{l} is the indecent light coming from the light source.

Specular reflection

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

- \mathbf{r} , which is the direction that a perfectly reflected ray of light would take from this point on the surface.
- \mathbf{v} , which is the direction pointing towards the viewer (such as a virtual camera).
- m , which is a shininess constant for this material, which is larger for surfaces that are smoother and more mirror-like. When this constant is large the specular highlight is small.

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

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

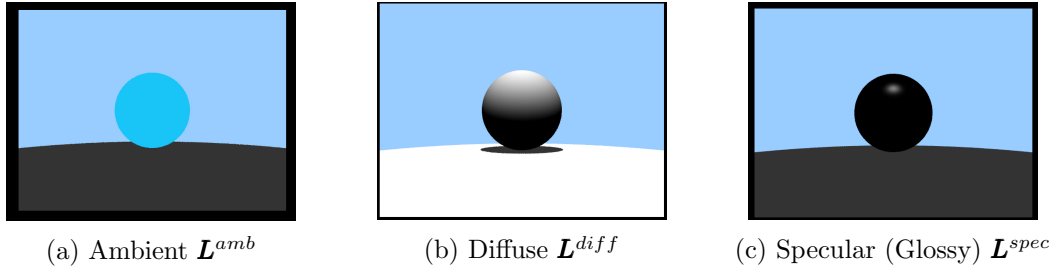


Figure 7: Visual illustration of the Phong equation

4 Performance

This is the most exciting part of the project; the previous chapters explained how to set up the current Raytracer and what methods have been used to implement it; however, the result is not shown yet. This chapter will apply different performance tests on the Raytracer to show its average performance without optimizing the datastructure.

In the renderer the next setup is used:

Key	Value
Height	480
Width	640
DataStructure	None
Anti-aliasing samples	2
PC info	Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz 1.99 GHz RAM 16.0 GB

Table 1: The Raytracer settings.

Three different simulations were used to test the performance and took their average for each scenario or model, as shown in Table 2. The more simulations are run, the more accurate results we get; however, three simulations are used to test the performance because each takes approximately one hour. As discussed before, performance is the main discussion point in this project rather than the accuracy of the renderer; hence the spent amount of time to render should be discussed and optimized. In order to render more than three thousand spheres, as shown in Table 2, this can take up to one hour; this is not optimal. One model only as the Bunny needs one hour to be rendered is not practical in real-life simulation applications. We need to find the most expensive block code the Raytracer spends most of its time optimizing the rendering. The used Raytracer can be divided into the following phases:

- **Creating a scene:** For this part *createScene()* method is responsible for creating the scene, this includes loading primitives from a .obj file, this steps is governed by only the number of primitives to be created this means it is worth optimizing it, because it depends on how big the scene is. It takes ≈ 1 minutes from the overall rendering time, which is $\approx 1\%$ of the total time, hence this step is not so dominant.
- **Ray casting:** This part usually contains two for loops to go over through all the pixels in the screen, which depends on the width and height of the image to be rendered. The smaller the size of the image, the faster the rendered become as the for loop gets smaller; however, this is not helpful as we want to have a high-quality image and not lose pixels. The time complexity of the basic algorithm is $O(whr)$, where w and h are the width and

the height of the display screen in pixels; and r is the complexity of shooting a ray into the scene. (reference to Accelerating algorithms for Ray Tracing) For each pixel, there is a ray to be cast to the scene, and then the next two sub-steps are applied:

- **Intersection tests:** Most objects in the scene are composed of a combination of basic shaped objects like spheres or triangles; these basic objects are connected in a mesh to build bigger complicated objects, and to render the model, each object has to be tested against the shooted ray, to test if it is visible to this ray or not. For each object, there is ***isIntersected*** function used to test the intersection. This function itself is not huge, but repeating it n times can be time-consuming.
- **Evaluate pixel color:** Evaluating the pixel's brightness and if it is transparent or reflects light, all these steps are done after the intersection test returns true, because if the rays intersect the surface object, we want to evaluate it is proper to color, not only the diffuse color. This needs a recursion operation to cast more and more rays into the scene, and this depends on the settings of the raytracer as the parameter usually called depth d .

This concludes that the worst case time complexity for shooting a ray is $O(nd)$, where n is the number of spheres and d is the maximum level of recursion established. For a non-stochastic raytracer, d is constant and given by the user; hence it is not interesting for now; hence with a fixed width, height, and depth, a naive raytracer depends on the number of objects in the scene only n .

- **Write pixels values into a file:**


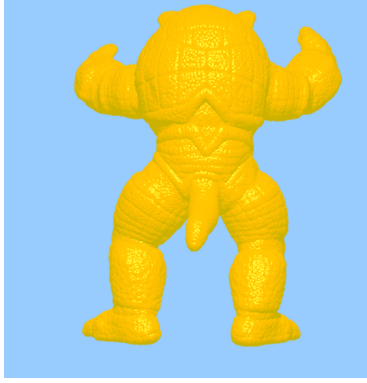
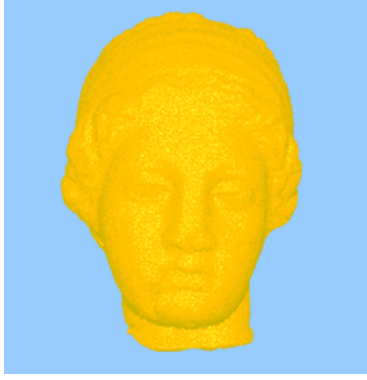
Model	Spheres	Sphere intersec- tion tests	Time taken to render	Result	
Bunny	35947	1841.98m	104.31 minutes		
Armadillo	49990	1822.94m	45.34 minutes		
Igea	134345	7278.88m	187.5 minutes		

Table 2: Performance comparison for different scenarios on 480X640.

5 Object subdivision

6 Spatial Subdivision

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

References