D.A.F.T.
Final Defense Report

Florian Lainé
Angela Saadé
David Wu
Timothy Edward Pearson

Contents

1 Introduction

Ray casting is a technique used to create 2D images and maps from 3D environments in computer graphics. This process is used to create a 2D representation of the 3D environment, and can be used for applications such as video games, architectural visualization, and robotic mapping. However, creating efficient and accurate ray casting algorithms can be challenging.

The DAFT Engine Library is a project of a real-time 3D engine designed to be easy to use and integrated into existing systems. In the final version, we aim to provide a library capable of rendering transparent objects with textures as well as reflections in an efficient maner. Eventually, we plan to add physic characteristics to objects and watch them interact in real time.

The project consists in making our own 3D engine starting from nothing. Indeed, our program will be capable to create 2D images from 3D environments in computer graphics. At the end, our project could be use as a C library for those who need a 3D engine to generate images for applications such as creating graphics for video games, architectural visualization or robotic mapping. Here are some examples of the most known 3d engine that are used nowadays.

Figure 1: Unity3D

Figure 2: Unreal

1.1 Group presentation

1.1.1 Global presentation

Our group is composed of 4 members : David WU, Florian Lainé, Angela Saadé and Timothy Edward Pearson. We are all students at EPITA for 2 years until now, and for this project, we wanted to create our own 3D engine, using the C programming language. Some of us are comfortable with this type of project, while it is the first time for others. Within the group, we can find various profiles : when some of us like programming, others might prefer doing mathematics or physics. With all our skills, we will try to carry out this project.

1.1.2 Timothy Edward Pearson

I am really looking forward to this project and I think that it will serve very useful in the future. Furthermore, building a 3D engine relies partially on vectors and linear algebra which we have done in maths class so it will be fun to apply these subjects to a real world situation. We intend to add some physics to the objects in our world if we have time which I am very excited about. I am eager to learn more physics application to my work at EPITA projects and better my knowledge in C programming.

1.1.3 Florian Laine

I'm really eager to start this project ! This 3D engine is a great opportunity of deeply understanding a lot of math and algorithm optimization at the same time. I introduced this project idea to my team mates and they immediately seemed to agreed. I personally am truly into the 3D processing since a few months ago and am super excited to start ! I have been doing some test on my own up to now but I am thrilled to take it to the next level by getting involved as part of a group work. I want to help as much as possible considering my few previous experiences and can feel a bit delighted about it. I obviously am very enthusiastic improving my skills and knowledge by implementing a whole new architecture and discovering new ways of solving the issues I can already have faced. Finally, this project represent for me a huge opportunity to accomplish one of my child dream I had when I was wondering how 3D graphics are working while playing some of my favorite games.

1.1.4 Angela Saade

I am excited to begin this project and see the potential it holds for me to gain a deeper understanding of 3D processing. This project is a great opportunity to learn, improve my skills by being part of this group. Even though I have had little to no experience with this subject and working in a team can be challenging sometimes, but I am determined to help and contribute as much as possible. It is a new type of project for me; building approximately everything from scratch, and it motivates me to take on this challenge.

1.1.5 David Wu

Being a novice at programming, this type of project seemed to be a real challenge for me. Indeed, I've never looked into such a project before : it includes loads of mathematical notions but also a great algorithm interest. To some extent, I am excited to begin the realization of this project, since I truly think that it can bring me some knowledge and skills that I need for my future engineer career. I know that I have teammates that I can count on, who will be able to help me if I struggle. Moreover, since I have been playing video games for several years, I have always been interested in 3D engines even if I have never had the opportunity to study the subject. I hope this project will make me discover a new exciting field of computer sciences.

1.2 Project presentation

The project is composed of several steps, the main stage being the stage of ray casting, including the computations that will allow us to render the objects on the screen, at the end of the program. In order to carry out this project, we built our own architecture that will be useful in each stage of our project. In addition, we have our own .obj file parser, which basically convert these files into something readable for our program. There are several ways to implement 3D engine, the one we chose being the Ray casting process, that will be explained in details later on. However, implementing an efficient and accurate ray casting process can be challenging. To proceed, we have used existing algorithms and tried to make them fit into our project. Finally, we have made our own architecture, our own parser for the .obj files, and our own way to implement ray casting.

2 Origin and Type of Project

2.1 Project's Origins

The idea of making a 3D engine came from one of us. Following consultation, we were all up to realize this project. Since each member of our group had already manipulated 3D engine such as Unity3D in the context of the S2 project, we could see which expectations were required for such a project. Moreover, we know that 3D engines are nowadays useful in several domain, such as video game conception, movie realization, but also in scientific fields, in order to simulate complex situations in a realistic 3D environment.

Afterwards, we found that this project could be a real challenge for us, since it includes a deep reflection on how to optimize our algorithms, this reflection being one of the foundations of today's programming. Furthermore, this project requires great understanding of 3D geometrical mathematics, what really made us want to look into it to improve our skills in this domain. Indeed, even if we have already manipulated software including 3D engine, most of us had never thought about how these engines were made.

2.2 Type of Project

As said before, our goal is to make our own 3D engine. This project will be entirely made in the C programming language. As a goal, our 3D engine will be able to generate 2D images from a 3D environment. The quality of our results will rely heavily on our ability to optimize the calculations given to the computer. Indeed, it exists several ways to make 3D engine nowadays. We chose the Ray Casting process, which will be explained later in the specification book. At the end, we would like the user to be able to place objects in a 3D environment, making it as realistic as possible. Therefore, we will have to implement functions that will be presented as "tools" for the user.

Also, our program will aim at returning the most realistic images as possible, (processing at 30 frames per second at minimum) which is equivalent to generate a smooth 3D video stream in real time. Finally, we want our project to include a whole API part, in order to make it easy handling for those who want to generate simple 3D videos. Indeed, we aim to make it easy for users to understand the different features implemented, in order to deliver the best possible user experience.

3 State of the Art

3.1 Simulate ray

Ray casting and ray tracing are both techniques used in computer graphics to simulate the behavior of light, but they are based on different principles and are used for different purposes.
Ray casting is a simpler and more efficient technique used for rendering 3D scenes from a 2D perspective. It works by casting rays from the camera, or viewpoint, into the scene and determining the closest object that the ray intersects. The color and other properties of that object are then used to determine the color of the pixel in the final image. Ray casting is mainly used for creating 3D games and real-time applications, it is fast and efficient but less accurate.
Ray tracing, on the other hand, is a more complex and computationally expensive technique used for creating highly realistic images. It works by tracing the path of light as it bounces off of and passes through objects in a scene. This allows for accurate simulations of reflections, refractions, and shadows, as well as global illumination and ambient occlusion. Ray tracing is mainly used for offline rendering, animation, and architectural visualization, it is accurate but takes more time to render.
In summary, ray casting is a fast, efficient method for creating 3D images from a 2D perspective, while ray tracing is a more accurate and computationally expensive method for creating highly realistic images.

3.1.1 Ray Tracing

Ray tracing is a technique used in computer graphics to generate images by simulating the behavior of light. It works by tracing the path of light as it travels through a 3D scene, taking into account the properties of the objects in the scene and how the light interacts with them.
The process of ray tracing starts by casting a ray from the virtual camera, or viewpoint, into the scene. The ray is tested for intersection with any objects in the scene, if an intersection is found, the algorithm calculates the color of the pixel based on the surface properties of the object, such as its texture, reflectivity, refractive index and so on.
It also takes into account other lighting conditions in the scene, such as the position and intensity of light sources, as well as the properties of the surrounding environment. This allows for the simulation of realistic reflections, refractions, and shadows, as well as global illumination and ambient occlusion.
Ray tracing is a more computationally intensive technique than other rendering methods, such as rasterization, but it can produce highly realistic images with accurate lighting and shading. It is mainly used in offline rendering, animation, and architectural visualization, and increasingly used in real-time applications such as video games and virtual reality.
Ray tracing is used in a variety of fields and industries. Some examples include:

- Film and animation: Many high-budget films and animated movies use ray tracing to create photorealistic visual effects.
- Gaming: Some video games, such as the popular game Minecraft RTX, use ray tracing to create more realistic lighting and shadows.

- Architecture and design: Architects and designers use ray tracing to create realistic visualizations of buildings and other structures, to help clients understand what the final product will look like.
- Automotive design: Automotive companies use ray tracing to visualize and test the aerodynamics of car designs.
- Medicine: Medical researchers use ray tracing to simulate the behavior of light in the human eye and other biological systems.
- Scientific visualization: Researchers in fields such as astrophysics and atmospheric science use ray tracing to create accurate visualizations of complex data.
- Virtual Reality: Ray tracing is used in Virtual Reality to create realistic and immersive experiences.

These are just a few examples, but ray tracing is used in many other fields as well.

3.1.2 Ray Casting

Ray casting is a technique used in computer graphics to generate images by tracing rays from the viewpoint or camera into the scene. It is used to determine which object in the scene is visible to the viewer and to calculate the color of each pixel.
The process of ray casting starts by casting a ray from the virtual camera, or viewpoint, into the scene. The ray is tested for intersection with any objects in the scene, if an intersection is found, the algorithm calculates the color of the pixel based on the surface properties of the object, such as its texture, reflectivity, refractive index and so on.
Unlike Ray tracing, Ray casting does not take into account more complex lighting conditions and interactions such as reflections, refractions, and global illumination. Therefore, the images generated by ray casting are generally less realistic and detailed than those generated by ray tracing.
Ray casting is a simpler and more efficient technique than ray tracing, which makes it suitable for real-time applications such as video games and interactive simulations. It is used to create 3D games and real-time applications, it is fast and efficient but less accurate than ray tracing.
Ray casting is used in a variety of fields and industries, some examples include:

- Gaming: Ray casting is widely used in video games to quickly and efficiently render 3D environments and characters. It is particularly useful for first-person shooters and other fast-paced games, as it can quickly determine which objects are visible to the player and calculate the appropriate colors and textures.
- Virtual Reality: Ray casting is used in VR and AR applications to quickly render 3D scenes and objects, allowing for smooth and responsive interactions.
- Medical imaging: Ray casting is used in medical imaging to create 3D visualizations of internal organs, bones, and other structures, making it easier to diagnose and treat various medical conditions.

- Robotics: Ray casting is used in robotics to map and navigate environments, allowing robots to safely and efficiently move around in the real world.
- Industrial design: Ray casting is used in industrial design to create 3D visualizations of products and machinery, allowing designers to test and refine their designs before they are built.
- Computer-aided design: Ray casting is used in CAD programs to quickly render 3D models and visualizing them.

These are just a few examples, but ray casting is used in many other fields as well.

3.2 Rasterization

Rasterization is a technique used in computer graphics to convert vector graphics and 3D models into a 2D image representation. It works by taking a 3D scene and dividing it into a series of small, rectangular pixels, called fragments or samples. The properties of each fragment, such as its color, depth, and texture, are determined by processing the scene's geometric data, and the final image is created by assembling the colored fragments on a 2D surface.
The process of rasterization starts by transforming the 3D coordinates of each vertex in the scene into 2D screen coordinates. Then, the fragments are generated by filling the area of the triangle formed by the three vertices of the 3D object. The fragments generated in this way are called "primitives". The color and other properties of each primitive are determined by applying textures, lighting, and shading.
Rasterization is a fast and efficient rendering technique, which makes it suitable for real-time applications such as video games, interactive simulations, and virtual reality. It is widely used in computer graphics because it is relatively simple and can be implemented on a wide variety of hardware platforms.
It's important to note that rasterization is not as accurate as ray tracing or ray casting, it is more suitable for real-time applications because of its speed, but images generated using rasterization are less realistic and detailed.

4 Goal and Interest of the Project

Creating a ray casting engine will allow us to delve deeper into the mathematical and computational principles of 3D graphics. We will gain a comprehensive understanding of how to simulate the behavior of rays in a 3D space, including how to determine the intersection of rays with 3D objects. This knowledge will be invaluable in understanding the fundamental principles of 3D graphics, which can be applied in a wide range of fields. Additionally, the process of creating a ray casting engine will give us hands-on experience in implementing and optimizing the algorithms used in 3D graphics, further strengthening our understanding of the underlying principles.

The project we are embarking on will be an exciting opportunity to not only create a dynamic 3D rendering engine, but also to develop our programming skills in the process. One of the key components of the project will be the implementation of the ray casting algorithm, which is used to generate the images we see on screen. This will require a deep understanding of the mathematics and physics behind the algorithm, as well as the ability to optimize it for real-time performance.
Additionally, we will be implementing backface culling and spatial partitioning techniques, which are essential for improving the performance and efficiency of the engine. These techniques will help us to optimize the rendering process by reducing the number of unnecessary calculations and improving the overall performance of the engine. Overall, this project will be a challenging and rewarding experience that will help us to become better programmers and gain a deeper understanding of 3D graphics.

As a group, we are highly interested in game development and creating a ray casting engine is a perfect opportunity for us to gain hands-on experience in real-time 3D game development. Through the process of creating this engine, we will gain a deeper understanding of the intricacies and challenges of game development, and will be able to apply this knowledge to our future endeavors. In addition to gaining valuable technical skills, we will also have the opportunity to develop our problem-solving and teamwork skills as we work together to create a functional engine.
Furthermore, once the engine is completed, we will have the potential to use it to create our own simple game. This would be an exciting opportunity to apply our knowledge and skills in a practical setting, and to see the fruits of our labor come to life in the form of a fully-functioning game. Additionally, this will give us a chance to test and optimize the engine in real-world scenarios, and identify any potential areas for improvement. Overall, creating a ray casting engine is not only a great learning opportunity, but it also has the potential to be the starting point of our own game development journey.

Finally, creating a 3D engine is challenging, we will need to debug and optimize the code, and troubleshoot any issues that arise. This will help us develop problem-solving skills and gain experience working as a team.

5 Progress

6 Division of tasks

6.1 Task Distribution

Tasks	Florian	Angela	David	Timothy
Website				
Ray Casting				
Optimisation				
OBJ File				
Physics				
Utils				
Textures				

Leader **Substitute**

6.2 Advancement planed for defenses

Advancement in percentage			
Tasks	Defense 1	Defense 2	Defense 3
Website	30%	60%	100%
Raycasting	50%	70%	100%
Optimisation	30%	50%	100%
OBJ files	80%	100%	100%
Utils	40%	60%	100%
Textures	0%	30%	100%

6.3 Real advancement

Advancement in percentage			
Tasks	Defense 1	Defense 2	Defense 3
Website	40%	90%	100%
Raycasting	80%	90%	100%
Optimisation	20%	70%	100%
OBJ files	80%	80%	100%
Utils	70%	100%	100%
Textures	0%	0%	100%

More than expected **Less than expected**

As you see on the tabular above, we have advanced more than expected on all of the fields except for the textures. Since we underestimated the optimization of the real time rendering, to implement moving meshes in real time would not give the results expected at this time. On the other hand, we have implemented shadows in our world. Instead of adding textures to the triangles, we have decided to allow the user to move around freely in real time at low quality (without reflections and shadows) and then take a snap of the world from the current cams position by pressing the P key.

The OBJ Files section (Obj parsing) is now working for obj files that include polygons with up to vertices allowing us to rending a wider range of obj files. The last feature to be implemented is parsing and applying the textures to the triangles.

7 Utils

7.1 Structure Tools

To initialize each of the structures, we have used functions that malloc a pointer for the corresponding structure and set the fields either with arguments or with default values.

Since we have used malloc, free functions are necessary for any structure that contains pointers to other structures. For any of the structures that include pointer lists, a loop frees each of them.

In order to add a vertex to a mesh, we have implemented a simple function that takes three floats and a mesh pointer. the same if done for a triangle however size_t arguments are used for index the vertices list instead of three floats. This has been done to follow the structure of obj files to ease parsing.

To add structures to the world, we have used a another simple function that checks if we have reached the maximum meshes/lights or cameras before adding them.

7.2 Architecture

After the restructure of the first version, we realised that a point structure was unnecessary and costly. Therefore, we place all the point structures by a float array for size 3.
All structure ids have been removed as they used too much space in memory and were only needed for very few functions.
We have also created a Sphere object. In order to have the smoothest looking shape, a mesh structure to imitate a sphere would require too many triangles. Therefore, the sphere object is simply a center point and a radius meaning that it would have to be rendered separately to the meshes.

7.3 Math Tools

Since we have created our own strictures, we had to create our own set of maths functions, only using functions like sqrt, cos and sin from the math.h library. Majority of these functions have a version that takes pointers as arguments and another that take static structures.

1. Dot Product
2. Cross Product
3. Scalar Product
4. Vector Addition
5. Vector subtraction
6. Normalize Vector
7. Norm of a Vector
8. Max and Min

8 Preprocessing

8.1 Scene Parser

A `.scene` file is a configuration file used to define the elements and properties of a 3D scene. Here's a documentation explaining how to use each line and its components:

Camera Line:

- Syntax: `camera x y z pitch yaw fov`
- Components:
  - `x`, `y`, `z`: The position coordinates of the camera in the scene.
  - `pitch`: The vertical rotation (in degrees) of the camera. Positive values tilt the camera downwards, and negative values tilt it upwards.
  - `yaw`: The horizontal rotation (in degrees) of the camera. Positive values rotate the camera to the right, and negative values rotate it to the left.
  - `fov` (optional): The field of view of the camera in degrees. It determines the extent of the scene visible through the camera. If not specified, the default value is 90.

Skybox Line:

- Syntax: `skybox path`
- Components:
  - `path`: The file path to the skybox texture. The skybox is a cube that surrounds the scene and provides the background appearance. The texture should be in a format compatible with the rendering engine.

Light Line:

- Syntax: `light x y z r g b`
- Components:
  - `x`, `y`, `z`: The position coordinates of the light source in the scene.
  - `r`, `g`, `b`: The RGB color values of the light. Each value should be between 0 and 1, representing the intensity of red, green, and blue, respectively.

Sphere Line:

- Syntax: `sphere x y z r material_path`
- Components:
  - `x`, `y`, `z`: The position coordinates of the sphere in the scene.
  - `r`: The radius of the sphere.

- `material_path`: The file path to the material properties of the sphere. The material defines how the sphere interacts with light, including its color, reflectivity, and other visual properties.

Mesh Line:

- Syntax: `mesh mesh_path x y z scale rot_x rot_y rot_z`
- Components:
  - `mesh_path`: The file path to the 3D mesh model that will be rendered in the scene. The mesh should be in a compatible format such as OBJ or FBX.
  - `x`, `y`, `z`: The position coordinates of the mesh in the scene.
  - `scale`: A scaling factor applied to the mesh. It determines the size of the mesh in the scene.
  - `rot_x`, `rot_y`, `rot_z`: The rotation angles (in degrees) around the X, Y, and Z axes, respectively. They define the orientation of the mesh in the scene.

Note: The components mentioned above should be replaced with appropriate values or paths relevant to your scene. Ensure that the file paths are correctly specified to load the required textures, materials, and mesh models.

By using these different line components in a `.scene` file, you can define and configure various objects and properties to create immersive 3D scenes.

8.2 Obj File Parser

An object file is a simple text file format commonly used for representing 3D geometry. And when implementing a 3D engine, using object files is essential, that's why a parser was implemented that retrieves relevant information from those files. They consist of a collection of data points, including information about vertices, texture coordinates, normals, and faces. In more details, vertices (v) are individual points in 3D space, defined by their x, y, and z coordinates. Texture coordinates (vt) define how a texture is applied to a surface, while normals (vn) describe the orientation of a surface at each point. Faces (f) are defined as a set of vertices that form a polygon or triangle. They are typically represented as a series of indices that reference the vertices, texture coordinates and normals.

Object files have a fixed format, but the number of vertices in a face can vary from 3 to a large number. A face with 3 components forms a triangle but a face with more than 3 components forms a polygon which will be then divided into triangles. The old version only works with 3 full components, as well as 3 components only with vertices. The current parser, loads object files with faces that have at most 5 components and implements the computation of triangles from a polygon.

This object parser reads the file at a given path for the first time and computes the total number of vertices and triangles. A face (f) can contain more than three components. So When a face (f) is encountered, the number of components should be computed in order to compute the correct number of triangles it forms; if it has

```
f 1715 1716 1690 1461
f 1713 1711 1467
```

Figure 3: Two faces one with 4 components and the second with 3

more than 3 components. Those two numbers are then going to be used to build a mesh. Then, the file is read for a second time line by line to store the scaled vertices' coordinates in the appropriate attributes in a three elements float array. After that, the vertex is added to the mesh. As mentioned above, faces in different object files may have more than 3 components. So in the second reading, when a face is encountered, the number of it's components is computed and at the same time the indexes of the vertices that form it are stored in an integer pointer (p) initialized to the elements. If the number of components is greater than 3, (p) is resized to the respective size of components. Afterwards, the integer pointer (p) is used to compute the triangles and add them to the mesh. In fact, an index i is used, which starts at 1 and goes at maximum to n-2, where n is the number of components that forms the face. In addition, an integer array of 3 elements is created to store the indices of the vertices that form the triangle in order to add the triangle to the mesh. The first element of the array is the value of the first element of p -1, the second is the value of the element at index i in p -1 and the third is the value of the element at index i+1 in p -1. Finally, when all the triangles are added to the mesh, the mesh is added to the world and the file is closed.

8.3 Material File Parser

Textures enhance the visual realism of 3D objects by adding surface details, colors, and patterns. The MTL parser enabled the extraction of material properties, from the MTL file, which allowed for the proper application of textures on objects during rendering. By accurately parsing and utilizing the MTL file, the parser ensured that the textures were applied with the correct mapping and alignment, resulting in realistic and visually appealing 3D scenes. Thus, the MTL parser played a vital role in achieving visually stunning and immersive 3D graphics by enabling the effective utilization of textures on objects.

A Material Template Library file is a plain text file that contains material definitions and associated properties for 3D objects in a 3D graphics application. It works in conjunction with Object files, where the OBJ file specifies the geometry of the 3D objects, while the MTL file defines the appearance and characteristics of the materials applied to those objects.

An MTL file typically consists of one or more material definitions, each represented by a series of properties. These properties describe various aspects of the material, including its color, texture, reflectivity, transparency, illumination model, and more. Here are some common properties found in an MTL file: -newmtl: specifies the start of a new material definition
-Ka: ambient color of the material

-Kd: diffuse color of the material
-Ks: specular color of the material
-Ns: shininess of the material
-Ni: index of refraction of the material
-d: dissolve factor of the material

These properties are specified for each material defined in the MTL file, allowing for fine-grained control over the appearance of 3D objects in the scene.

By using the material definitions and properties specified in the MTL file, our 3D rendering engine can accurately render and shade objects, providing realistic textures and visual effects in the rendered scene.
The main objectives were to extract material properties such as ambient color, diffuse color, specular color,, transparency in the MTL file. The MTL File was opened and its content was read line by line, identifying different material definitions and associated properties. Regular expressions techniques were used to extract material properties. The extracted properties were stored in a suitable material structure. In addition, some changes were made in other parts like in the process of adding the triangles to the mesh and while loading the obj files. In more details, after storing the respective components in the material structure, all the materials were stored in a pointer so we can have acces to them easily and the number of loaded materials is stored in another pointer. Also, some changes were added to the loading of obj files. Generally, in obj files that are related to a material, there is a line at the top of the file "mtllib -name of a material file-" and this specifies to which mtl file this object is related to. In addition, there is a "usemtl -name of material-" which specifies which material in the mtl file should be applied to the faces that are after this line.

```
typedef struct {
    float color[3];
    float ambient[3];
    float diffuse[3];
    float specular[3];
    float shininess;
    float reflection;
    char name[256];
    SDL_Surface* texture;
} material;
```

Figure 4: Material Structure

In addition, to complete the material, we also have textures mapping that are complementary to those materials. Instead of finding lines beginning with Ka, Ks, Kd... we find lines starting with :

```
-map_Ks  Specular texture
-map_Ka  Ambiant texture
-map_Kd  Diffuse texture
-map_Ke  Reflection texture
```

You can find below the mtl file corresponding to a R2-D2 modelization:

```
# Blender MTL File: 'None'
# Material Count: 1

newmtl R2D2_IlluminationMaterial
Ns 96.078431
Ka 0.000000 0.000000 0.000000
Kd 0.690196 0.690196 0.690196
Ks 0.001176 0.001176 0.001176
Ni 1.000000
d 1.000000
illum 2
map_Kd R2D2_Diffuse.jpg
map_Ke R2D2_Reflection.jpg
map_Ks R2D2_Specular.jpg
```

You can see below the diffuse and specular textures from the R2D2 mtl file:

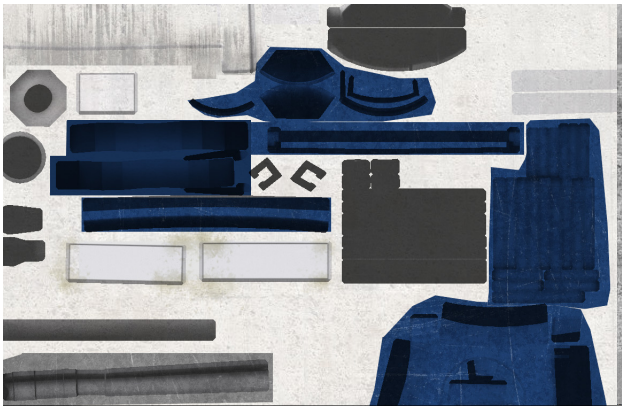


Figure 5: R2D2Diffuse.jpg

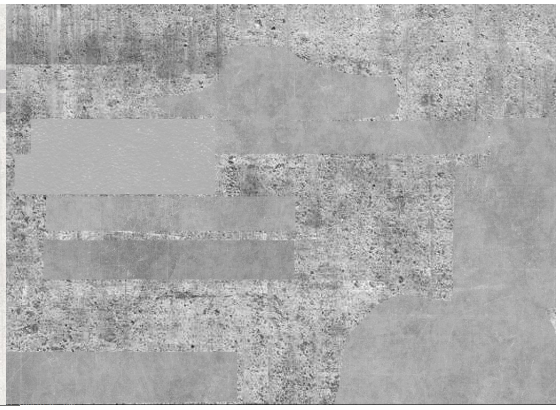


Figure 6: R2D2Specular.jpg

Figure 7: Material and texture applied on R2D2 object

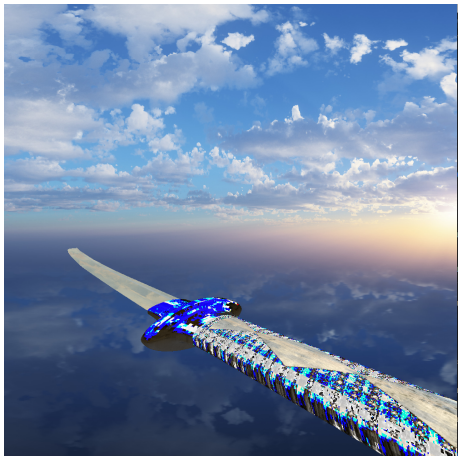


Figure 8: Material applied on katana object

9 Rendering

9.1 Ray-Triangle Intersection

We consider that each of the objects we are rendering are composed of a meshes that contains polygon faces or spheres with a center point and radius. Triangles are the simplest polygon leaving us an easy and optimized algorithm to calculate the ray intersection. The ray intersection function is used to determine if a ray has hit a triangle in the environment. Since each ray represents a pixel, the pixel color depends on which triangle it it intersects.
Our ray intersection function is inspired from the Möller-Trumbore intersection algorithm. Here are the two main structure we will use to implement ray triangle intersection :

```
typedef struct triangle
{
»···size_t vert[3];
»···point normal;
}triangle;
```

Figure 9: Triangle structure

```
typedef struct ray
{
»···point pos;
»···point dir;

»···int hit;
»···point contact;
»···triangle **collision;
}ray;
```

Figure 10: Ray structure

The first step to determine whether the ray intersects the plane that a particular triangle lies on or not. We can cancel out any rays that are parallel to the plane using the dot product between the ray direction and the result a vector subtraction between two vertices of the triangle. Therefore, we need to get the edges for the triangle considered as vectors. Considering a triangle with points A(x_A,y_A,z_A), B(x_B,y_B,z_B) and C(x_C,y_C,z_C), here are the formulas that allows us to determine the edges AB and AC:

$$\vec{AB} = (x_B - x_A, y_B - y_A, z_B - z_A)$$

$$\vec{AC} = (x_C - x_A, y_C - y_A, z_C - z_A)$$

We use arbitrarily the vector $\vec{AC}$ to compute the following cross product with the direction vector of the ray. Once done, a scalar product is computed with the resulted vector and the vector $\vec{AB}$.

$$\text{Cross product (h = edge1 X ray\_dir) :}$$
$$\vec{H} = (AC_y * dir_z - AC_z * dir_y, AC_z * dir_x - AC_x * dir_z, AC_x * dir_y - AC_y * dir_x)$$

$$\text{Dot product (a = edge2 . h): } a = (AB_x * H_x, AB_y * H_y, AB_x * z_A)$$

The result, a, will allow us to determine if the ray is parallel to the plane formed by the triangle. We set a constant called EPSILON to 0.0000001 and see if the absolute value is less than our constant. In the case that it is, we consider the ray parallel to the plane and stop the calculation there.

From there we use the Barycentric co-ordinates of the current triangle to find out if the intersection is within the triangle.

The point of intersection can be written as: $P = wv_1 + uv_2 + vv_3$

The coefficients must be non-negative and sum to 1, so w can be replaced with $1 - u - v$

so we have:
$P = (1 - u - v)v_1 + uv_2 + vv_3$
$P = v_1 + u(v_2 - v_1) + v(v_3 - v_1)$
The ray equation can be written as $\vec{r}(t) = O + t\vec{v}$
Equating the two gives :

$$O + tD = v_1 + u(v_2 - v_1) + v(v_3 - v_1)$$
$$O - v_1 = -tD + u(v_2 - v_1) + v(v_3 - v_1)$$

which is equivalent to:

$$\begin{bmatrix} | & | & | \\ -d & (v2 - v1) & (v3 - v1) \\ | & | & | \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - v_1$$

Then we use Cramers rule to solve for t, u and v.
Finally, if t is negative the ray doesnt intersect.
If u or v is not within the interval [0,1] and the um of u and v does not equal one, then the point is not within the triangle.

For the sphere, we project the vector (sphere_center - ray_pos) onto the direction vector of the ray which will give us the t value of the ray/ with that we have the closest position of the ray to the sphere and check if this value is less than or equal to the radius of the sphere.

9.2 Skybox

A skybox is a cube-mapped texture that is used to represent the background of a 3D scene in computer graphics. The purpose of a skybox is to provide a realistic representation of the sky and surrounding environment in a scene, while reducing the computational overhead of tracing rays for distant objects.

In ray tracing, a skybox is implemented as a large cube that completely encloses the scene. The interior of the cube is mapped to a single texture, which represents the sky and surrounding environment. When a ray misses all of the objects in the scene, it is assumed to have hit the skybox, and the color of the skybox is used as the background color for that pixel.

Figure 11: Skybox example

To map a ray to the skybox texture, UV mapping is used to determine the location of the ray on the skybox image. UV mapping is a technique for mapping a 3D position to a 2D texture coordinate. In the case of a skybox, the UV mapping is done by projecting the ray direction onto the faces of the skybox cube and finding the corresponding pixel on the skybox image.

for any P point on the sphere of $radius = 1$, calculate d the unit vector from P to the sphere origin. Assuming that the sphere's poles are aligned with the Y axis, UV coordinates in the range $[0, 1]$ can then be calculated as follows:

$u = 0.5 + \frac{arctan(d_z, d_x)}{2\pi}$
$v = 0.5\frac{arcsin(d_y)}{\pi}$

We can now compute the color of the pixel and then, fill the empty pixels of the screen by a nice Skybox !

In conclusion, a skybox is a cube-mapped texture that is used to represent the background of a 3D scene in computer graphics. UV mapping is used to map a ray to the skybox texture, and the formulas for u and v can be calculated using the arctangent and arcsine functions. The use of a skybox can greatly improve the performance of ray tracing by reducing the computational overhead of tracing rays for distant objects.

9.3 Lights and shadow

9.4 Lights

For this defence, we used the lights to make a very simple render for our shadows. The lights structure has been defined for the last defence and is implemented as follows:

As described above, the lights have three fields that are explicit : color, intensity and a position that acts as a vector. For further features, we might add direction field

```
typedef struct light
{
    float pos[3];
    float color[3];
    float intensity;
}light;
```

Figure 12: Light structure

to the lights in order to make shadows more realistic. In the world structure, we have implemented a linked list of lights, so that it is simpler to iterate over it, especially for the rendering of the shadows. Implementing lighting in a 3D engine project involves calculating the ambient, diffuse, and specular reflection of light sources. To begin, the ambient light component provides a base level of illumination to all objects in the scene. This is achieved by adding a global ambient light color to the final color calculation of each object, creating a uniform illumination across the scene.

Next, the diffuse lighting component is computed by determining how the light interacts with the object's surface. The direction of the light source and the object's normal vector are utilized to calculate the intensity of the diffuse reflection. By applying the Lambertian lighting model, which assumes that light is scattered equally in all directions on a diffuse surface, the resulting intensity is multiplied by the material's diffuse color.

Finally, the specular reflection accounts for the shiny highlights on the object's surface. This is computed using the viewer's position, the light's direction, and the object's normal vector. The reflection vector is calculated and compared with the viewer's vector to determine the intensity of the specular highlight. The material's specular color is then multiplied by this intensity to obtain the final specular reflection.

By combining the ambient, diffuse, and specular components, the overall lighting effect can be achieved for each object in the 3D scene. This process is repeated for each light source present in the scene, with their contributions accumulated to create the final lighting result. Implementing these calculations accurately and efficiently enhances the visual realism of the 3D engine, allowing for realistic illumination and captivating virtual environments.

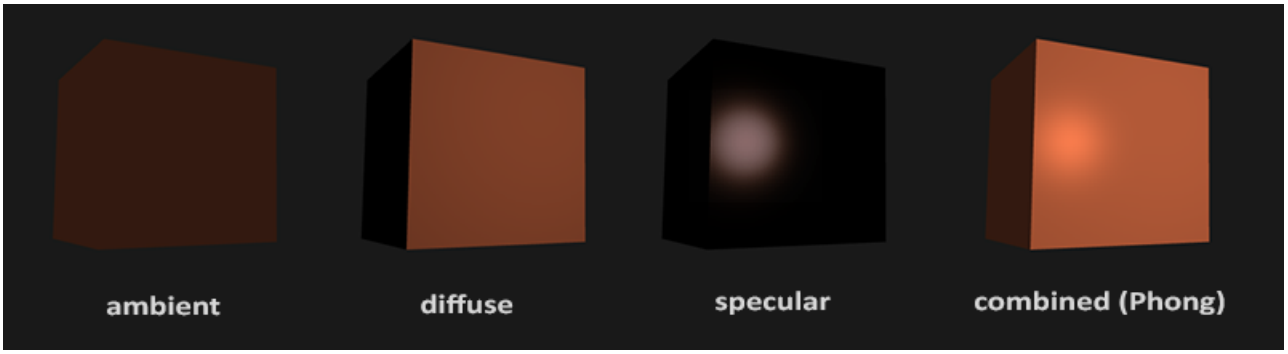


Figure 13: Lights rendering on a cube

9.5 Shadows

Since we are using the ray casting process for the whole project, it is quite simple and efficient to render shadow in the frame. We wanted to add a basic rendering of shadows for all our objects present in the environment. For now, we were able to render objects in the environment such as simple polygons, but also more complex forms like cows, human shape meshes... You can find below the render of the shadow that we have achieved for this defence.

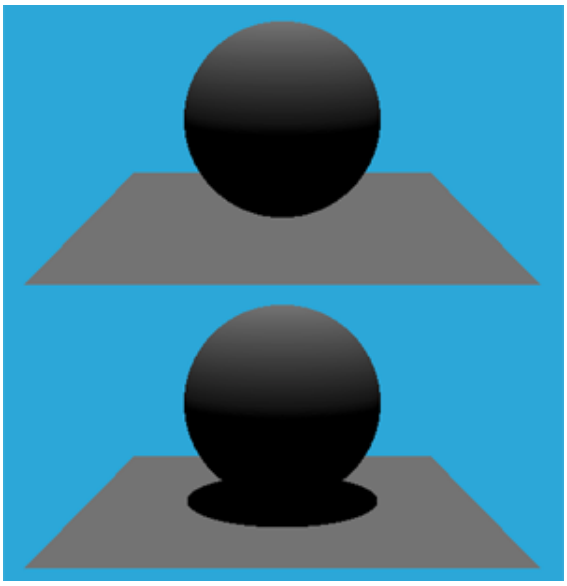


Figure 14: First shadow rendering

In the adjacent image, we can see a diffuse sphere and a diffuse plane illuminated by a directional light source. What's missing now in this image to make it more realistic is the shadow of the sphere on the plane. Fortunately, simulating the action of objects casting shadows on over objects in the scene is pretty straightforward to simulate in CG. Well, it depends on the algorithm used to solve the visibility problem. Since we use ray tracing, computing shadowing can be done while the main image is being rendered (it doesn't require precomputing a special image such as a shadow map).
All we need to do is cast a ray from the object visible through a particular pixel of the frame, from the point of intersection to the light source. If this ray which we call a shadow ray intersects an object on its way to the light, then the point that we are shading is in the shadow of that object.
To make it easier to understand, we can do a simple comparison with the real world and illustrate it with a simple image.

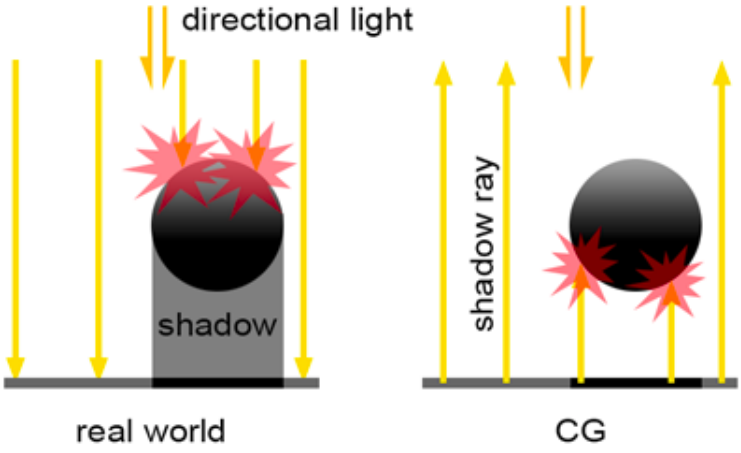


Figure 15: Shadow rendering in CG

For instance, in the real world, light travels from the light source to the eye. Shadows are the consequence of some objects along the light path blocking the light from reaching another surface that is located further along the way. Though, in CG (Computer Graphics), it is more efficient to trace the path of light from the eye to the light source. Thus, we first trace a camera or primary ray, then find an intersection point (the surface that is visible for the pixel from which the primary ray was cast) and then trace a shadow ray from that point of intersection to the light. The difference between what happens in nature, and how we compute shadows in CG is shown in figure 8. To get the intersection point with the meshes, we have a structure named rayresult which contains the result of every ray (Figure 9).

```
typedef struct ray_result
{

    float mint;
    triangle* tri;
    mesh* m;
    float normal[3];
    float color[3];
    float reflectivity;

} ray_result;
```

Figure 16: Ray result structure

We will use this structure to store the first intersection point with the closest mesh from the ray launched at the position of the camera. Once we had identified the closest object, we cast shadow rays from the object's surface point to the light source. If the shadow ray intersected with another object in the scene before reaching the light source, we concluded that the point on the object's surface was in shadow. Before we launch the ray to the light, the "mint" field is set to the distance between the light and the point so that we can check if this distance changed after the launch. Indeed, the mint field indicates the distance between the origin of the ray and the first object that has been hit. If the mint value changes, it means that we have hit an object before the light and therefore the pixel at the origin is set to black.

9.6 Further optimizations

Note that when we compute whether a point is in the shadow of an object, the order in which the objects intersected by the shadow ray doesn't matter. It doesn't matter if the object that the shadow ray intersects is not the closest object to the ray's origin. All we care about in this case of a shadow ray is to know if the ray intersects an object at all in which case the point from which the shadow ray was cast is in shadow. Practically, this means that for shadow rays, we could very well return from the trace function as soon as we intersect one object in the scene. This can potentially save some computation time.

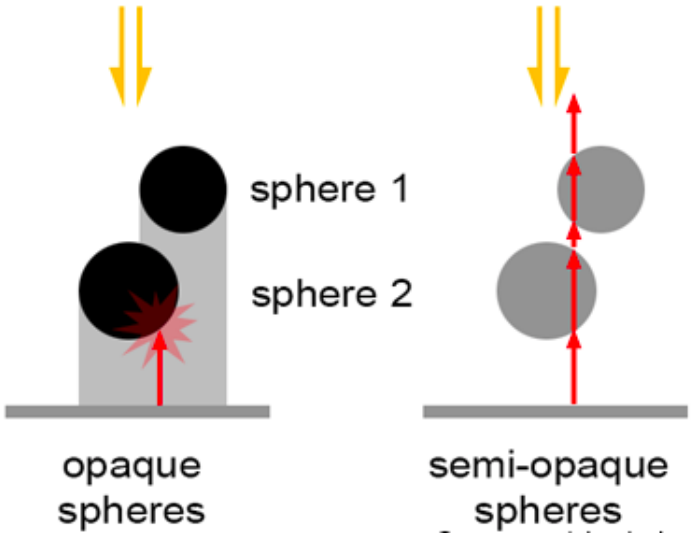


Figure 17: Illustration for optimization

Though this optimization is only possible if all the objects in the scene are fully opaque. If some of the objects are transparent, then light rays that are potentially being attenuated while traveling through these semi-opaque objects should keep traversing the objects until they either encounter an opaque object on their way to the light or reach the light itself (only in the case of spherical lights, not in the case of distant light which is assumed to be located at infinity). In this section, we will not show how to handle semi-opaque objects for shadow rays. Trying to optimize the code for shadow ray if you plan to support semi-opaque objects, later on, is not worth the pain but is left as an exercise if you wish to.

9.7 Loading bar

A loading bar appears on the screen when the key "p" is pressed. It represents the progress of copying the lower-quality image into a higher-quality image. It visually indicates how much of the copy has been completed in green and how much is left in black.

9.8 Rasterization

When rasterizing a sphere onto a flat screen, you first need to define the dimensions of the screen, specifying its width and height in pixels. This will determine the resolution of the output image. Next, you should determine the center coordinates and radius of the sphere you want to rasterize.

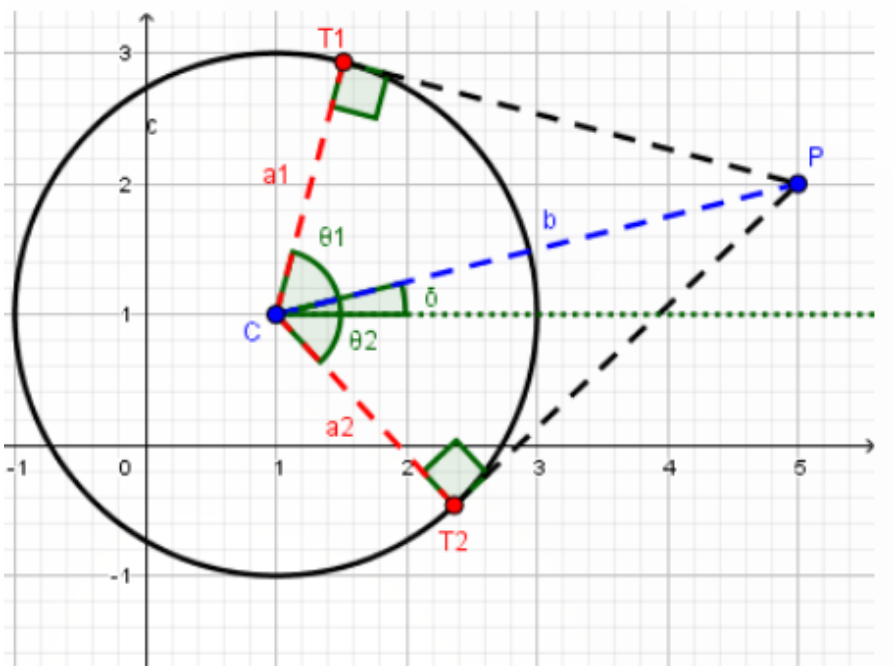
To begin the rasterization process, you iterate over each pixel on the screen. For every pixel, you perform a calculation to determine its corresponding position on the sphere. This involves converting the 2D pixel coordinates back into 3D coordinates on the sphere using a process called reverse projection.

By normalizing the screen coordinates to a range of [-1, 1] based on the screen dimensions, you can calculate the corresponding x, y, and z coordinates on the sphere. The z-coordinate is determined using the equation of a sphere, and the x and y coordinates are scaled by the radius to obtain the final coordinates on the sphere.

Once you have the 3D coordinates for each pixel, you can determine whether the pixel falls inside or outside the sphere. This is done by calculating the distance between the sphere's center and the pixel using the distance formula. If the distance is less than or equal to the sphere's radius, the pixel is considered inside the sphere; otherwise, it is outside.

Based on this determination, you assign a color to each pixel. Pixels inside the sphere can be colored to represent the surface of the sphere, while pixels outside can be given a background color or left empty to represent the absence of the sphere.

Since we know the radius and center on the screen, we only need to check pixels that are within a bounding box of the circle on the screen.



For all other objects, project their bounding boxes points on the the screen and then take the max in the x and the y axis to get all the pixels contained in an axis aligned square according the the screen dimensions.

10 Video

Looking towards the upcoming defense, we are also excited to unveil another new feature in our 3D engine: the ability to capture and display multiple high-quality images within our virtual environment. Our goal is to allow users to capture images from different angles and perspectives, and then display them in sequence to create a video-like experience that showcases the details and features of our virtual environment. To achieve this, we plan to capture multiple high-quality images at different points in the virtual world. This new feature will allow us to showcase the richness and complexity of our virtual world and provide a more interactive experience for users. We believe it will be a significant addition to our presentation at the upcoming defense. To do this, we will allow user input to set point in our world in which we will move the camera on using interpolation.
For this feature, we have mainly used the FFmpeg technology, that is a powerful tool for this purpose. Indeed, FFmpeg is a powerful and widely used open-source software suite for handling multimedia data. It serves as a command-line tool that allows users to manipulate, convert, and stream audio and video files. The name "FFmpeg" stands for "Fast Forward MPEG" since it was initially developed to handle MPEG video formats. However, over time, FFmpeg has expanded its capabilities and now supports a vast range of multimedia formats, codecs, and protocols.

FFmpeg provides a comprehensive set of tools and libraries for tasks such as video encoding, decoding, transcoding, filtering, and streaming. It supports a wide array of video and audio codecs, allowing users to convert media files between different formats. Additionally, FFmpeg offers various filtering options to modify and enhance multimedia content, including resizing, cropping, adding watermarks, and applying special effects.

One of the key strengths of FFmpeg is its cross-platform nature, as it can be used on multiple operating systems such as Windows, macOS, and various Linux distributions. Its command-line interface allows for automation and scripting, making it popular among developers and system administrators.

Furthermore, FFmpeg's flexibility and extensibility have led to its integration into numerous applications, media players, and frameworks. Its libraries, such as libavcodec and libavformat, provide a programming interface that developers can utilize to incorporate FFmpeg's capabilities into their own software projects.

Overall, FFmpeg is a versatile and powerful multimedia framework that enables users to manipulate, convert, and process audio and video files with efficiency and flexibility. Its widespread adoption and active development community make it a go-to tool for many multimedia-related tasks.

Here is the main reasons why FFmpeg is considered efficient for video processing:
- Broad Format Support: FFmpeg supports a wide range of video and audio formats, including popular ones like MP4, AVI, MKV, and more. This broad format support allows users to handle and convert videos in different formats without needing to install multiple software tools.

- High Performance: FFmpeg is designed to be fast and efficient, utilizing optimized algorithms and libraries for encoding, decoding, and processing multimedia data. It leverages hardware acceleration when available, such as GPU acceleration, to further enhance performance and speed up video processing tasks.

- Multithreading and Parallel Processing: FFmpeg can take advantage of multi-core processors by utilizing multithreading and parallel processing techniques. This allows for concurrent execution of tasks, enabling faster video encoding, decoding, and transcoding operations.

- Streaming Capabilities: FFmpeg excels in streaming multimedia content over various protocols and network environments. It supports both input and output streaming, making it a reliable tool for live streaming, video-on-demand (VOD) streaming, and video conferencing applications.

- Extensive Filters and Effects: FFmpeg provides a rich set of video filters and effects that can be applied during video processing. These filters allow users to resize, crop, rotate, add text or watermarks, adjust colors, apply special effects, and perform other manipulations on the video stream. This flexibility enables users to enhance and customize videos according to their specific requirements.

- Command-Line Interface: FFmpeg's command-line interface offers flexibility and automation. Users can create scripts and batch processes to perform complex video operations, allowing for efficient and consistent video processing workflows.

- Open-Source and Community-Driven: FFmpeg is an open-source project with a large and active community of developers. This means that the software is continuously improved, updated, and maintained by a dedicated community. Bugs are quickly addressed, and new features are regularly added, ensuring that FFmpeg remains a reliable and efficient tool for video processing.

Due to its performance, versatility, and extensive features, FFmpeg has become a popular choice for professionals and enthusiasts involved in video editing, transcoding, streaming, and various other multimedia-related tasks.
In the figure below, you can see white points, that corresponds to "checkpoint" for the output video. Indeed, during the process, the camera will go through each point and link the images frame by frame to create the video.

11 Website

The goal was to create a visually appealing and responsive website. The website needed to be accessible across different devices and screen sizes. To achieve this, we chose to leverage the power of Bootstrap, a widely-used framework known for its flexibilty and efficiency.

11.1 Bootstrap

Bootstrap is a powerful and widely adopted front-end framework that has revolutionized website development. Developed by Twitter, Bootstrap offers a comprehensive collection of pre-designed templates, CSS styles, and JavaScript components, making it an indispensable tool for developers. At its core, Bootstrap is designed to provide a responsive and mobile-first approach to web design, ensuring that websites are optimized for various screen sizes and devices. The framework's key advantage lies in its ease of use and flexibility. It provides a robust grid system that simplifies the layout structure of a website, allowing developers to create visually appealing and well-organized interfaces. Additionally, Bootstrap's extensive library of components, including navigation bars, buttons, forms, modals, and carousels, offers a wide range of ready-to-use elements that can be easily customized and integrated into a project. This allows developers to save time and effort by leveraging the pre-built components rather than building them from scratch. Another noteworthy aspect of Bootstrap is its focus on cross-browser compatibility, ensuring that websites built with the framework are compatible with different web browsers, minimizing the need for extensive testing and troubleshooting. Moreover, Bootstrap's responsive design capabilities and consistent styling ensure a cohesive and professional appearance across various devices and platforms. With a large and active community, Bootstrap continues to evolve, introducing new features, enhancements, and responsive design patterns to keep up with the latest trends in web development. In summary, Bootstrap is a versatile and user-friendly framework that empowers developers to create modern, responsive, and visually appealing websites efficiently. Its extensive library of components, responsive grid system, cross-browser compatibility, and active community support make it an invaluable tool for web development projects of all sizes and complexities.

Advantages of Bootstrap:

-Responsive Design: Bootstrap offers a mobile-first approach, enabling the website to automatically adjust its layout based on the user's device, whether it's a desktop, tablet, or smartphone.

-Cross-Browser Compatibility: Bootstrap ensures that the website appears consistent and functions properly across different web browsers, saving developers from the hassle of extensive browser testing.

-Customization Options: Bootstrap can be customized to match the specific design requirements of a project. Developers can choose to use the default styles or modify

them using CSS to create a unique visual identity for the website.

We used Bootstrap to achieve the following:
-Layout Design: We employed the Bootstrap grid system to structure the website's layout, ensuring that the content was displayed optimally across various devices. This involved dividing the page into responsive rows and columns to create a flexible and balanced structure.

-Styling and Theming: By leveraging Bootstrap's CSS classes and components, We applied consistent styling throughout the website. This included typography, buttons, forms, and navigation elements.

-Responsiveness: We utilized Bootstrap's responsive utilities to create a fluid and adaptable design. This allowed the website to resize and reposition elements automatically based on the user's screen size, enhancing the overall user experience.

By leveraging Bootstrap, we successfully developed a dynamic and responsive website for the project. The framework's powerful features, such as its grid system, extensive component library, and customization options, enabled us to create a visually appealing website. Bootstrap proved to be an invaluable tool in achieving a consistent and responsive design across different devices, ultimately enhancing the overall user experience.

When the user is on our website, he can find a brief project presentation. He can find also the link to the projects' github repository and the link to the documentation of the project. When scrolling down, he can see the timeline of the project. Then, he can visualize some example of renderings: a sphere in a skybox with the reflection applied to it, mutltiple spheres, an iron mask and a character. Then he can see the group presentation as well as the dowloadable files as well as our book of specification. On the documentation link, the user can find the architecture of the project, the engine step by step, the errors encountered and different links and sources.
The user will find a detailed structure of the projects architecture with the explanation of each file and function, the usage and the errors are mentionned as well.

12 Optimisation

12.1 Bounding Boxes

Bounding boxes serve as a cornerstone optimization technique in computer graphics, particularly in the context of ray tracing, where performance is crucial. These spatial data structures act as efficient containers that enclose groups of objects within a defined box-like region. By encompassing objects with bounding boxes, complex scenes can be organized and processed more effectively.

In ray tracing, the intersection test between rays and objects is a computationally expensive operation. However, by leveraging bounding boxes, the number of these tests can be significantly reduced. When a ray traverses a scene, it first checks for intersection with the bounding box of an object. If there is no intersection with the bounding box, the entire object can be disregarded, saving precious processing time. This optimization technique greatly reduces the overall number of intersection tests required, resulting in faster rendering times and improved performance.

The impact of bounding boxes on performance becomes especially pronounced when rendering scenes with numerous objects. Without bounding boxes, each ray would need to be tested against every object in the scene, resulting in a significant computational burden. However, by utilizing bounding boxes, the number of actual intersection tests is dramatically reduced, often by orders of magnitude. This reduction in computational overhead allows for smoother real-time rendering, interactive applications, and the ability to handle more complex scenes.

Moreover, bounding boxes play a vital role in accelerating data structures used in ray tracing, such as kd-trees and BVHs. These structures organize objects hierarchically, based on their bounding boxes. This hierarchical arrangement optimizes the traversal process by reducing the number of objects that need to be tested for intersection with a given ray. By traversing the hierarchy and selectively testing only relevant objects, significant performance improvements are achieved. This results in efficient ray-object intersection calculations and further enhances the overall rendering speed.

In summary, bounding boxes are a crucial optimization technique in ray tracing, offering substantial benefits in terms of performance and efficiency. By reducing the number of intersection tests, bounding boxes enable faster rendering times, making real-time or near-real-time rendering achievable for complex scenes. When combined with acceleration structures like kd-trees or BVHs, bounding boxes further enhance the performance of ray tracing algorithms, resulting in efficient traversal and improved overall rendering speed. The utilization of bounding boxes represents a fundamental approach to achieving high-quality, real-time computer graphics rendering.

12.2 K-d trees

For effective spatial searches in multi-dimensional data, such as in the 3D rendering of meshes, a data structure known as a k-d tree, or short for k-dimensional tree, is

frequently employed.

A k-d tree can be used to speed up the ray-tracing procedure, which involves figuring out which objects in a scene intersect with a specific ray, while displaying meshes. The renderer can concentrate only on the necessary parts of the mesh by swiftly eliminating huge regions of the mesh that do not cross with the ray by arranging the triangles of the mesh into a k-d tree.

What makes a K-d tree more optimal than the Bounding Volumes is that instead of checking the intersection of the ray with a bounding box, we can check the ray intersection with a min and max value in a certain axis. This speeds up the traversal time.

Furthermore, only needing to spilt on dimension at a time increase the building time of the tree structure.

After the first creation of the K-d Tree, we found that the

12.3 Parallel Programming

Parallel computing is a critical factor in the context of ray tracing, as it enables substantial performance improvements in the rendering process. Ray tracing involves computationally intensive tasks, such as casting numerous rays through a 3D scene to simulate the complex interactions of light with objects and surfaces. By leveraging parallel computing techniques, the computational workload of tracing these rays can be effectively divided among multiple processors or cores, leading to significantly faster rendering times and enhanced efficiency.

One widely adopted approach to introducing parallelism in ray tracing is to partition the image into smaller tiles and assign each tile to a separate processor or core. This allows for concurrent processing of rays within each tile, harnessing the power of parallel execution. As the processors work independently on their assigned tiles, the ray tracing computations can proceed simultaneously, exploiting the parallel capabilities of the underlying hardware. Once all the tiles have been processed, the resulting partial images can be efficiently combined to generate the final rendered image. This approach effectively reduces the overall rendering time, especially for scenes with complex geometry and lighting effects.

In practice, various parallel computing frameworks and libraries can be employed to facilitate the implementation of parallelism in ray tracing. One popular example is the OpenMP library, which provides a straightforward and portable interface for adding parallelism to C, C++, and Fortran programs. By utilizing OpenMP directives, developers can annotate specific sections of the ray tracing code to indicate the regions that can be executed in parallel. These directives allow for automatic workload distribution and thread management, enabling efficient utilization of multiple processors or cores.

Moreover, parallel computing is not limited to just the ray tracing computations. It can also be leveraged in other stages of the rendering pipeline, such as intersection testing, shading, and post-processing. By parallelizing these tasks, the overall rendering process can be further accelerated, leading to real-time or near-real-time rendering capabilities.

In conclusion, parallel computing is a crucial factor in the optimization of ray tracing. By effectively dividing the computational workload and leveraging the power of multiple processors or cores, parallel computing significantly enhances rendering perfor-

mance and enables the handling of more complex scenes. Strategies like tiling the image and utilizing parallel computing libraries such as OpenMP offer efficient solutions for implementing parallelism in ray tracing. Overall, parallel computing techniques contribute to pushing the boundaries of realistic and interactive computer graphics by enabling faster and more efficient rendering.

13 Features

13.1 Further Optimisations

13.1.1 Optimized TLAS rotations

One of the way that can be used to optimize our implementation is using the TLAS rotation. TLAS instances can be more efficient for tracing if their local bounding boxes are axis-aligned. Even if we have no control over individual objects, we can come up with a global transform for all instances to improve tracing time.

However, instead of proceeding transform on each of the instances in the environment, we thought that rotating the ray for each objects instead of the objects themself could be an efficient solution to reduce the computation time and therefore maximize the optimization. Though, this method may not be beneficial for every type of environment : it is mainly used in environment where most objects are oriented in the same axis.

13.1.2 Alpha tested geometry

Alpha testing is a technique where fragments of a 3D model are discarded based on the alpha value of the texture. By using alpha tested geometry, the ray casting process can be improved because only the visible parts of the model are checked for intersection with the ray, saving computation time and improving performance. This is because the fragments that are not visible, or have a zero alpha value, are discarded before the intersection test, reducing the number of unnecessary calculations.
One way to reduce the tracing cost is to start with pre-tracing, where you trace local neighborhoods for each pixel in screen space, similar to screen space shadows or ambient occlusion techniques. You don't have to perform this step for every pixel on-screen to make sure that there is no additional overhead. For that, you can use additional data stored in G-buffer surfaces to mark the pixels that belong to hair or fur.

Diffuse global illumination or reflections on rough surfaces may not require precise albedo or alpha testing results. You can store averaged material values per primitive. This way, good results can be achieved by stochastically evaluating opacity during any-hit shader execution without the need for additional per-vertex attributes fetching and interpolation or texture sampling.

13.1.3 Denoising

Denoising is a technique used in computer graphics to remove noise or visual artifacts from a rendered image. In the context of ray tracing, noise often appears as random speckles or fluctuations in color and brightness, particularly in areas of an image that are poorly lit or have a lot of reflective surfaces. These artifacts can make the image appear grainy and lower the overall quality of the rendering.

Denoising works by analyzing the pattern of the noise in an image and then removing it by smoothing out the speckles. This results in a cleaner, smoother image with improved visual quality.

There are several different denoising algorithms that can be used, each with its own strengths and weaknesses. Some popular denoising methods include non-local means, bilateral filtering, and deep learning-based approaches.

Using denoising can have a significant impact on the performance of ray tracing. By removing noise from the image, fewer samples are needed to achieve a desired level of visual quality. This means that the rendering process can be completed faster and with less computational resources, leading to improved performance. Additionally, denoising can also help to reduce the amount of memory required to store the rendered image, further improving performance.

In summary, denoising is a technique that can help to improve the performance of ray tracing by reducing noise and visual artifacts in the rendered image, leading to faster rendering times and lower memory usage.

13.2 Image processing

Figure 18: Cow mesh with skybox

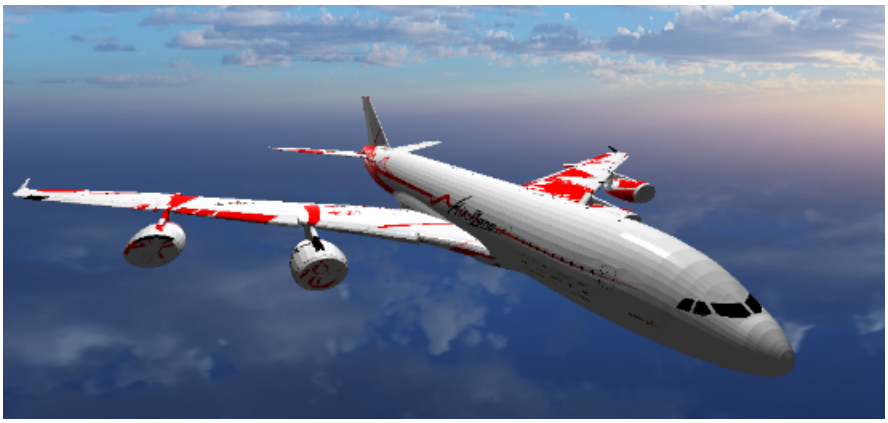


Figure 19: Plane with materials and textures

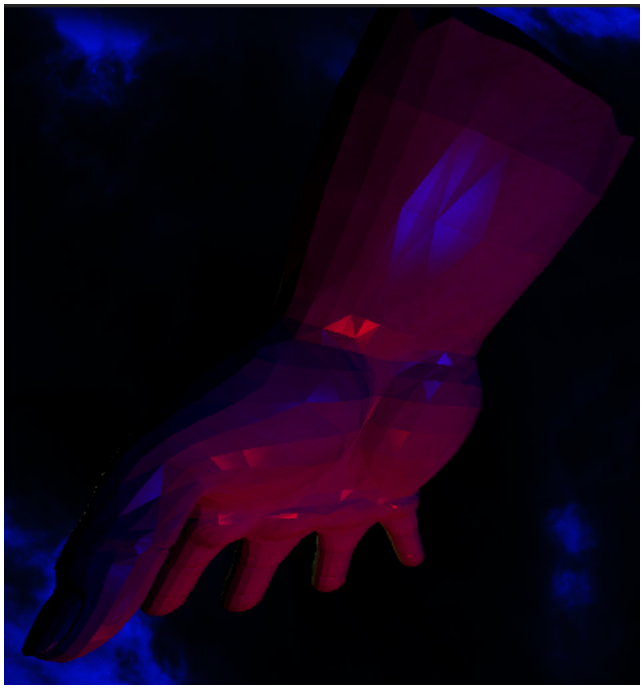


Figure 20: Hand with materials and textures

14 Conclusion

In summary, our project is focused on the development of a 3D engine that uses ray casting to produce highly detailed and realistic images and animations. Ray casting provides an effective way to determine the correct colors for objects in a virtual world and enables the creation of 2D images and maps from 3D environments. Our objective is to create a fully functional 3D engine that can be utilized for a broad range of applications such as video games, architectural visualization, and robotic mapping. In addition to the original scope of the project, we have implemented additional features including reflective objects and reflections, transparency, and an OBJ parser that can load more than three components. Furthermore, we have updated the website to

provide a more comprehensive overview of our engine's capabilities. We are confident that the use of ray casting in our engine will provide an efficient and powerful tool for producing lifelike images. We are excited to demonstrate the full capabilities of our engine during the final defense.

The journey of developing a 3D engine has been an incredible experience for all members of the group, both personally and professionally. As we come to the end of this project, we find ourselves filled with a profound sense of achievement and growth. Through countless hours of coding, debugging, and problem-solving, we have gained invaluable technical expertise and a deep understanding of computer graphics. Moreover, the project has fostered a strong sense of camaraderie and collaboration within our team. We have learned to work together, support one another, and overcome obstacles as a cohesive unit. The joy of witnessing our engine come to life, rendering stunning visuals and interactive environments, has been truly exhilarating. This project has ignited a passion for game development and computer graphics within each of us, leaving us inspired and eager to explore further possibilities. Overall, the 3D engine development project has brought us not only technical proficiency but also a deep sense of fulfillment and a shared bond that will continue to fuel our pursuit of innovative projects in the future.