Université de Montréal

# High-Performance Asteroid Field Rendering

by
Alexandre Marques Dias

Department of Computer Science
Faculty of Arts and Sciences

Master of Science (M.Sc.) Internship Report
presented to Dr. Pierre Poulin

April 2025

## ABSTRACT

This report details my internship at *Cosmographic Software LLC*, where I worked as a graphics programmer to implement a variety of GPU-based graphics optimisations for the software *SpaceEngine,* a real-time space simulation software. These optimisations can affect many aspects in the engine. They were implemented with the primary intent to render asteroid fields on screen. The main project of this internship was to implement the rendering of millions of instantiated asteroids in real time with little to no loss in visual fidelity. To accomplish that, I applied the following computer graphics optimization techniques: GPU-based per instance frustum culling with indirect draw, level of detail, and volumetric billboarding. The latter is based on a rotating texture atlas depending on the asteroid's position and rotation and the camera's position, rotation, and orientation. This resulted in the successful rendering of millions of asteroids at 60 frames per second. The implemented techniques have all been programmed from scratch in Vulkan. They are abstracted into the engine to be used for other scenarios in the future, such as rendering debris on the surface of planets.

**Keywords:**
**Graphics programming, GPU optimization, Vulkan, Real-time rendering, Compute Shaders, Culling, Billboarding, Texture Atlas**

## RÉSUMÉ

Ce rapport détaille mon stage chez *Cosmographic Software LLC*, où j'ai travaillé comme programmeur graphique pour mettre en œuvre une variété d'optimisations graphiques basées sur le GPU dans le logiciel *SpaceEngine*, un logiciel de simulation spatiale en temps réel. Ces techniques d'optimisation peuvent affecter de nombreux aspects dans le moteur. Elles ont été mises en œuvre avec l'intention de faire le rendu de champs d'astéroïdes à l'écran. Le projet principal de ce stage était de mettre en œuvre le rendu de millions d'astéroïdes instanciés en temps réel avec peu ou pas de perte de fidélité visuelle. Pour y parvenir, j'ai appliqué les techniques d'optimisation graphique suivantes : élimination d'instances hors pyramide basée sur GPU avec *indirect draw*, niveaux de détails, et le *billboarding* volumétrique. Ce dernier se met à jour sur un atlas de texture rotatif en fonction de la position et de la rotation de l'astéroïde et de la position, de la rotation et de l'orientation de la caméra. Cela a permis de rendre avec succès des millions d'astéroïdes à 60 images par seconde. Les techniques mises en œuvre ont toutes été programmées à partir de zéro dans Vulkan. Elles sont abstraites dans le moteur pour être utilisées pour d'autres scénarios dans le futur, par exemple, le rendu des débris à la surface des planètes.

**Mot Clés:**
**Infographie, optimisation GPU, Vulkan, rendu en temps réel, *Compute Shaders*, élimination, *Billboarding*, atlas de textures.**

# CONTENTS

The structure of this report is organized into five chapters. Chapter 1 introduces *Cosmographic Software LLC*, outlines its main project *SpaceEngine*, and describes the internship objectives and context. Chapter 2 provides essential background information on various computer graphics optimization techniques relevant to the internship, including frustum culling, GPU-based frustum culling, level of detail (LOD) for meshes, billboarding, textures, and texture atlases. Chapter 3 presents the detailed modeling and implementation of my solutions, covering the GPU instancing data structure, GPU-based frustum culling, the level-of-detail system, and the special billboarding system that I built to simulate 3D rotation animations on a 2D billboard. Chapter 4 goes into the results, discussing performance, visuals and comparisons of related projects. Finally, Chapter 5 concludes the report by summarizing achievements of the internship and discussing possible future work.

# CHAPTER 1

# INTRODUCTION

*Cosmographic Software LLC* is a company specialized in the development of real-time simulation and visualization of astronomy and astrophysics. At the time of this report, the company is based in the United States in New Haven, Connecticut. It counts fifteen full-time employees. Founded in 2022, *Cosmographic Software* is owned by Vladimir Romanyuk, a Russian astronomer and computer graphics programmer, and managed by Alexander T. Long, acting as Chief Operating Officer of the company. Although the company was founded recently, *Cosmographic*'s main project, *SpaceEngine,* has been in development by Vladimir since before 2010.

The main project of the company is a real-time space simulation software called *SpaceEngine* (Figure 1). It is a 3D astronomical visualization software that allows users to explore and navigate through the universe in real time, with a focus on scientific accuracy and attention to details. *SpaceEngine* allows users to explore real scenarios, such as the Solar System, or hypothetical scenarios, such as procedurally generated star systems. It can also customize celestial objects such as planets. It features hundreds of thousands of real celestial objects registered in astronomical catalogs, such as from the *Hipparcos* catalog [1] and from the *New General Catalogue of Nebulae and Clusters of Stars* [2]. Each of these real objects are placed accurately where they exist in a universe, the size of over 32.6 billion light-years on each side, centered on the barycenter of the Solar System. They are visually represented as scientifically realistic as possible. Besides these real objects, *SpaceEngine* also makes use of realistic procedural generation based on real scientific data to generate most of its universe, which results in extremely accurate real-time simulation of space. *Cosmographic* employs three physicists with various roles to ensure the scientific accuracy of the software.

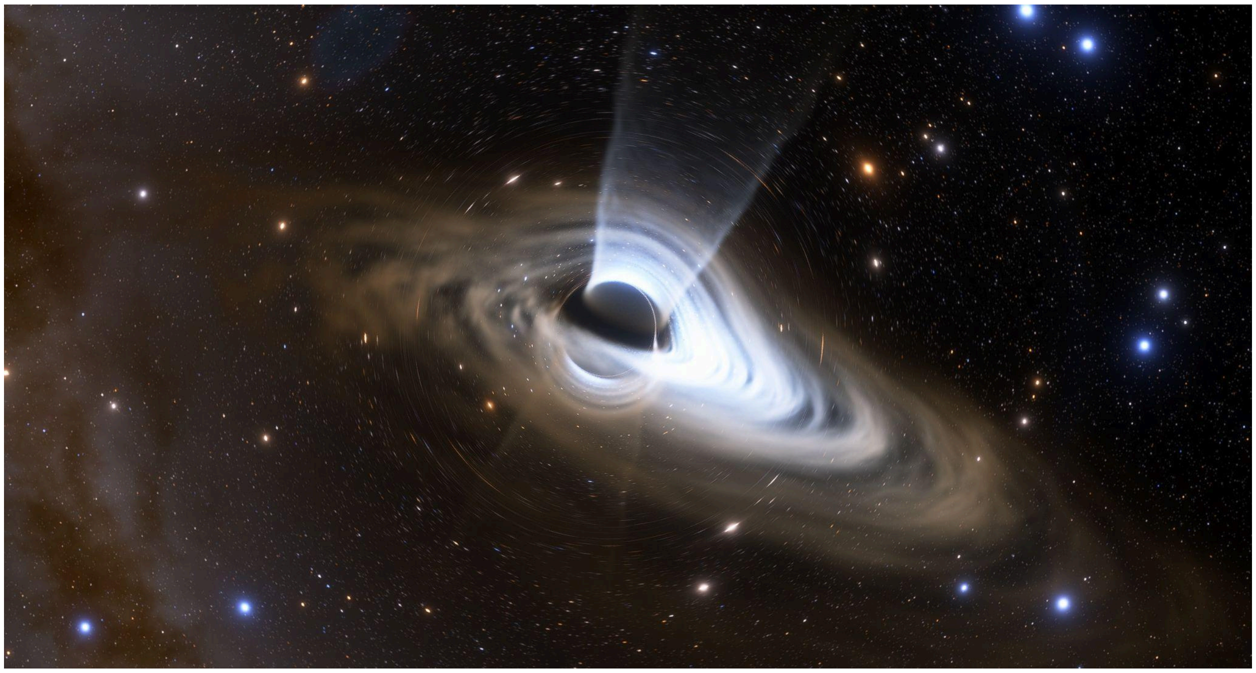


**Figure 1:** *SpaceEngine'*s General Relativity Visualisation.

The *SpaceEngine* development team is composed of three graphics programmers and two general programmers. Of  the two graphics programmers, Vladimir, the owner of the company, also acts as an authority for accurate space simulation because of his astrophysics background. Peter Ohlmann, my internship supervisor, is a talented graphics programmer with over 30 years of development experience in the video game industry while being an expert in Vulkan graphics.

The internship lasted ten months, from March to December 2024. My project was to efficiently render millions of asteroids on screen by combining multiple computer graphics techniques, and to develop the Vulkan infrastructure needed for the task and potential future ones. The main tools of the project were C++ for general programming, GLSL for shader programming, Vulkan for the graphics API, RenderDoc and Imgui for debugging.

# CHAPTER 2

# BACKGROUND

In this section, we discuss several common techniques that are used to optimize rendering performance and that were used during this internship. We start with frustum culling, a classic technique to discards objects outside the camera's view pyramid in order to reduce unnecessary drawing. Next, we discuss GPU-based frustum culling, a modern alternative where the culling work is offloaded to the GPU to better leverage parallel processing. Afterwards, we discuss mesh level of detail (LOD) of meshes, a method to dynamically adjust the number of vertices count based on camera distance to save on computational costs. We then go over billboards, an optimization technique about displaying objects as textures on a 2D rectangle. Finally, we look at textures, a fundamental concept in computer graphics, and then at texture atlases, which combine multiple textures into a single one to reduce the overhead of texture binding and draw calls.

## 2.1 Frustum Culling

Frustum Culling is a common object culling technique used in computer graphics to improve rendering performance. The main objective of frustum culling is to avoid rendering or instantiating objects that are outside of the view pyramid. The majority of frustum culling techniques are implemented on the CPU to avoid sending useless processing to the GPU before a 3D scene is loaded into it. This is done first by constructing the camera's view frustum, which is a portion of a solid like a cone or truncated pyramid that represents the volume of vision of a camera. It is made of six planes named from the camera's point of view : left, right, top, bottom with a near plane and a far plane to limit rendering distance.

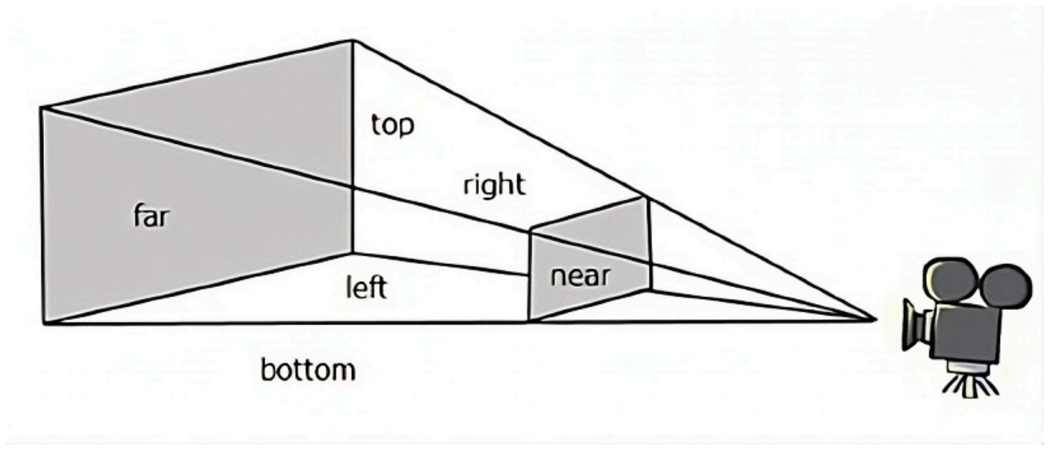


**Figure 2:** View Frustum.

The most common way to check if an object is inside or outside the view frustum is to test its bounding volume against each of the frustum's infinite planes. It can be done, for example, by taking the dot product of the plane's normal with the center of the bounding volume of the object, which will return us the distance the object is along the normal. We can use this result to determine which side of the plane the object lies on. If the normals of the planes point inward, and the result of a dot product between one of the planes is smaller than the negative distance to the center of the bounding volume, then it would mean the object is outside of the view frustum and it can be culled. It is also common to test all eight vertices of the bounding volume to provide more precise results at the cost of efficiency.

## 2.2 GPU-based Frustum Culling

An alternative and more modern technique for implementing frustum culling is called GPU-based Frustum Culling, also called Compute Shader Culling or GPU Instance Culling. It is a technique that forwards culling computations to the GPU rather than on the CPU, using compute shaders and indirect rendering. The main advantage of doing it this way is to benefit from the huge parallelism potential of compute shaders which can process many objects at once without sending data back and forth between CPU and GPU. This is done by having a list of object data stored on the GPU buffer, and a compute shader to cull them before the vertex shading phase. The compute shader algorithm to check if an object is within the limits of a frustum pyramid is similar to any typical implementation, the difference is that, instead of iterating over each object one at a time on the CPU, the GPU processes all objects in parallel. This can be done with bounding volumes or using only the object's center position, provided that they are spherical enough. Then, this is where indirect rendering comes in. Instead of sending individual draw calls from the CPU, the GPU handles it. The compute shader writes draw commands into a buffer (usually a DrawIndirect or DrawIndexedIndirect buffer). Each command contains all the information needed to render a mesh: the number of vertices, indices, the instance count, and offsets into the mesh and material buffers. The graphics pipeline then consumes this buffer directly with a single DrawIndirect or DrawIndexedIndirect call.

## 2.3 Level Of Detail (LOD) of meshes

Level Of Detail of meshes, often abbreviated LOD, is a technique used to optimize rendering performance by adjusting the complexity of a mesh based on its distance from the camera. The basic principle is that objects further away from the camera do not need as many details because they appear smaller on screen, so lower-resolution versions of those meshes can be used to reduce the amount of computations needed.

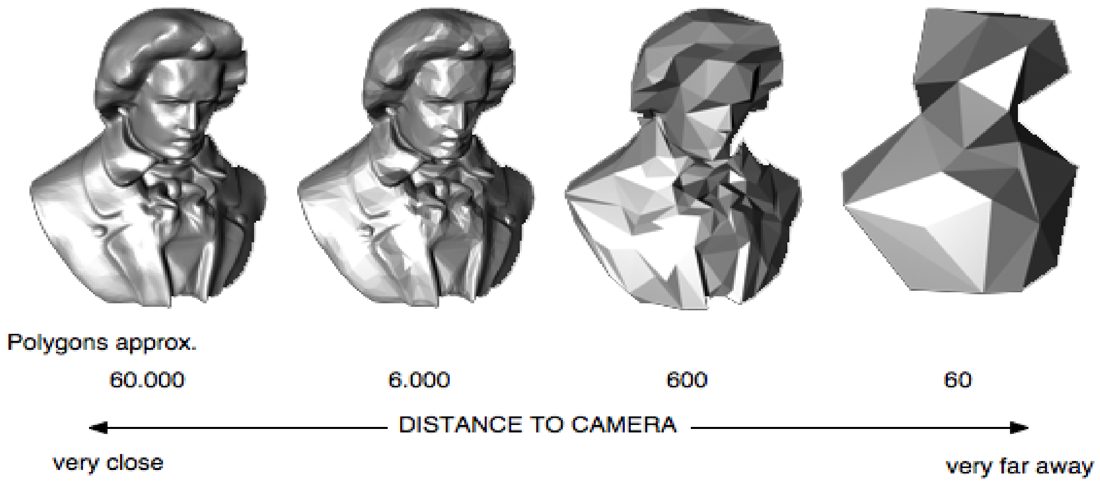


**Figure 3:** Level Of Detail.

A typical LOD system involves creating several versions of the same model, each with a different polygon count. High-resolution meshes are displayed when the object is close to the camera, while lower-resolution meshes are substituted in as the object moves further away. The transition between LODs is usually based on distance thresholds, although more advanced systems might factor in screen-space size or other metrics. If transition between LOD models is too apparent on screen, Hughes Hoppe's progressive meshes [3] provides a solution with its encoding of a continuous LOD representation with smooth, seemingly continuous transitions, at the cost of performance.

## 2.4 Billboards

Billboards are a rendering technique to display 2D quadrilaterals that are used to represent objects in a 3D scene, usually for objects like trees, particles, or distant objects. Instead of rendering complex 3D geometry, a billboard is a flat, two-dimensional rectangle with a texture mapped onto it, often with transparency. Billboards are usually implemented in a way such that their orientation always remains perpendicular to the viewer to display their texture, maintaining the illusion of volume despite being flat. Because they only require four vertices, billboards are computationally very efficient to render.

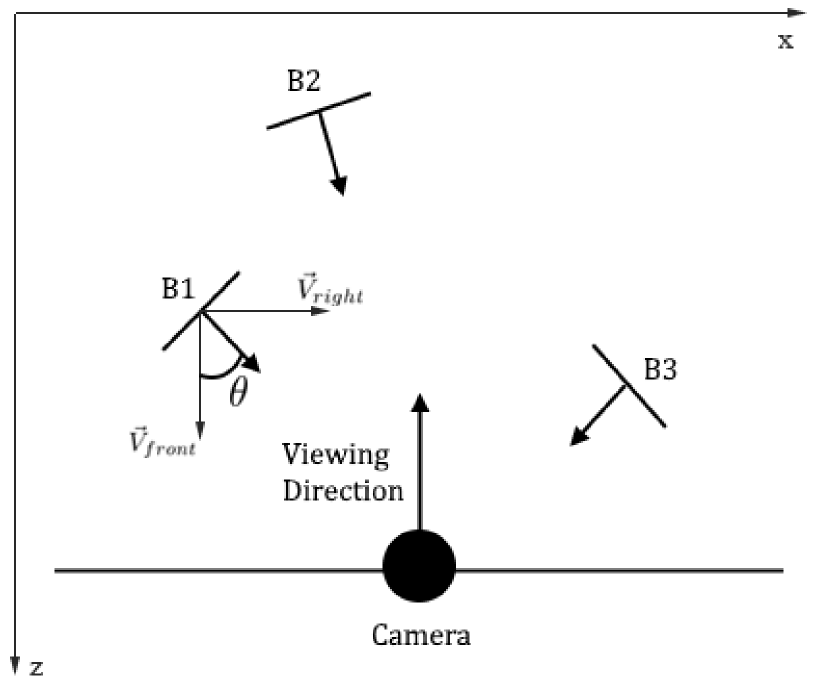


**Figure 4:** Billboards.

## 2.5 Textures

Textures are 2D images applied to the surfaces of 3D models to add color. They are typically mapped onto the surface of an object using UV coordinates. That tells the shader how to wrap the 2D image around the 3D shape. There are different types of textures, with different purposes, to name a few: diffuse or albedo textures define base colors, normal maps simulate surface bumps by perturbing normal vectors on the 3D model, and specular or roughness maps define how shiny or matte a surface appears. Textures are stored in GPU memory and sampled during rendering, very often in the fragment shader, to determine the final appearance of each pixel on the screen.

## 2.6 Texture Atlas

A texture atlas is a single large texture that contains a collection of smaller textures, often referred to as sub-textures or sprites. Instead of binding and switching between multiple textures during rendering, a texture atlas allows objects to sample from different regions of the same texture. This is often used to reduce the number of texture binds and draw calls, which improves rendering performance. Each sub-texture within an atlas is mapped to specific UV coordinates in a shader (often the fragment shader), which knows which part of the atlas to sample from for a given object and a given view direction. Texture atlases are commonly used for concepts like UI elements, animated 2D billboards, or different material variations on objects.

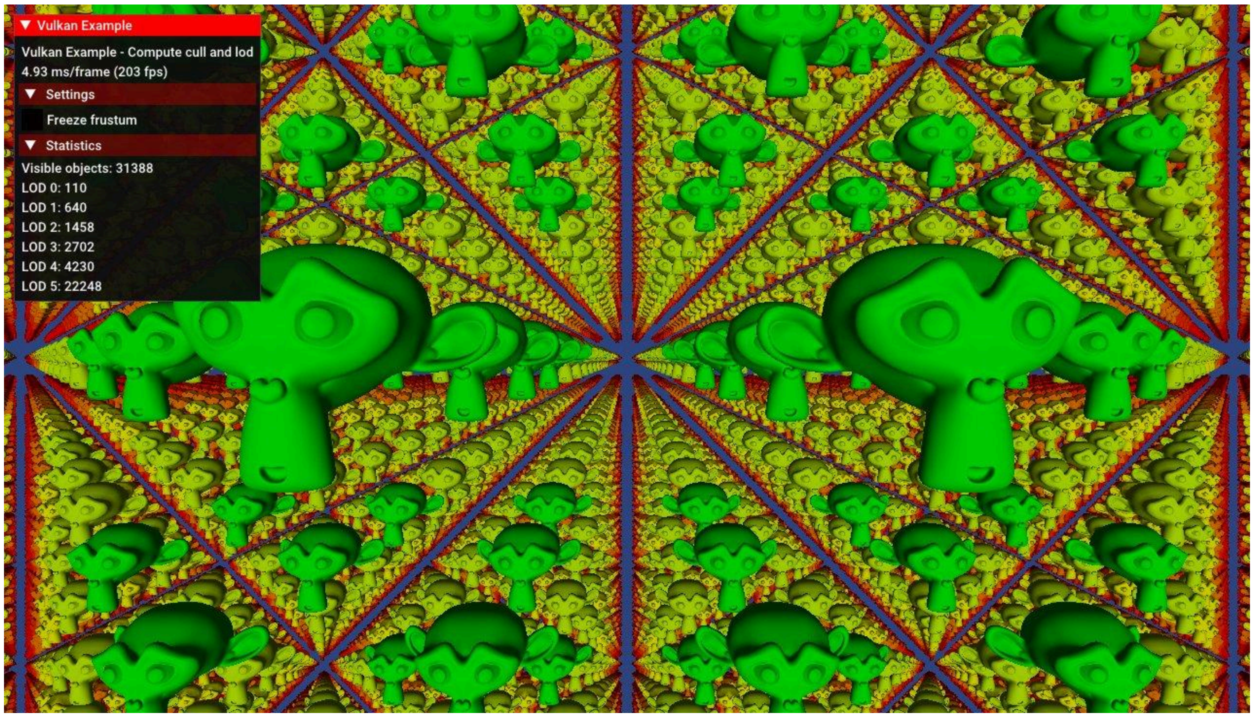
# CHAPTER 3

# MODELLING

In this chapter, we will go through the detailed modeling of the solutions implemented to successfully render millions of asteroids in real time. The chapter has eight sections, detailing the most important steps for this project. We start with an introduction describing the problem we need to solve and some context surrounding it. Then for the first three sections we detail how we implemented our frustum culling solution. First, we see how the compute shader was set up in the engine, then in a second section we see how we organised the asteroids' data on the GPU and finally, on a third section we look at the main frustum culling algorithm inside our compute shader to conclude the GPU-based frustum culling technique. Afterwards, we have a section on our LOD template structure and we explain how we define our distance function for choosing the correct LOD for a given asteroid. Then in the following section, we jump into billboards that represent asteroids at the lowest level of detail. Then in the next section we explain how we implemented our texture atlas and finally for the section, we explain how we animated our billboards using the texture atlas.

## 3.1 Introduction

The main problem we were trying to solve in this project is the ability to render millions of realistic asteroids on screen within at least 60 frames per second on mid-range computer hardware. Before my internship, there had been a previous attempt by another graphics programmer at solving this problem with the use of ray marching. The idea was to procedurally generate organic shapes with the use of signed distance functions (SDF) in screen space, entirely in a fragment shader. However, this process required too many computations per pixel, and could not reliably generate millions of asteroids on screen without a heavy loss of performance. Additionally, because the asteroids were all generated in the fragment shader phase of the graphics pipeline, they could not be distinguished as individual objects in our engine. This meant the asteroids could not interact with the player using various features present in our engine, such as clicking on an astronomical object to open an information window about it. Although the ray marching solution was mathematically complex, its advantage was that it was structurally simple, the majority of its code being contained in a single fragment shader. But in the end, this solution was simply not efficient enough, so we needed something else. We took inspiration from one of Shasha Willems's Vulkan projects. For context, Shasha Willems is a *Khronos Group* developer and advisory panel member. He is known for being one of the most respected Vulkan experts. He provided multiple tutorials and example projects implementing a variety of real-time computer graphic techniques using the Vulkan API. One of which was using computer shaders to implement GPU-based frustum culling to render tens of thousands of *Suzanne* objects with the help of distance-based levels of detail (see Figure 5). This was a good start for our project. However, we needed to render millions of asteroids with more polygons than a *Suzanne* model, more variety (multiple different asteroids), and within an engine that takes up additional processes (user interface, player controls, post processing, etc.). So additional rendering optimization techniques were needed, such as our billboarding system which we will discuss in detail later in this chapter.



```
▼ Vulkan Example
Vulkan Example - Compute cull and lod
4.93 ms/frame (203 fps)
▼ Settings
Freeze frustum
▼ Statistics
Visible objects: 31388
LOD 0: 110
LOD 1: 640
LOD 2: 1458
LOD 3: 2702
LOD 4: 4230
LOD 5: 22248
```

**Figure 5:** Sasha's Vulkan example with multiple instances of the Suzanne geometric model.

## 3.2 Compute Shader Structure

Before this internship, our project did not support any compute shader, and so, all the structures and pipelines in our Vulkan engine had to be built from scratch. Once that was done, our very first compute shader was written to optimize our asteroid fields by doing view frustum culling on the GPU. To make it as optimal as possible, the compute shader executes prior to the vertex and fragment shaders in the graphics pipeline, optimizing the rendering process by culling unnecessary geometry before any kind of rasterization occurs. This significantly reduces the workload for subsequent shader stages by discarding asteroids not visible within the camera's view. It is made especially efficient because we are executing the culling code for each asteroid in parallel on the GPU. Indeed, we defined the compute shader in a way so that it dispatches one compute invocation per asteroid instance.

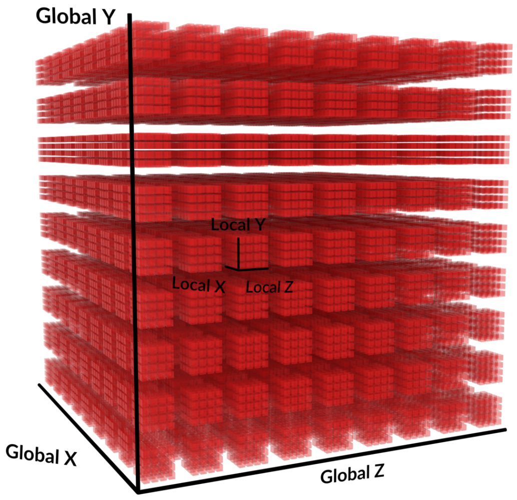


**Figure 6:** A work group split by its local size (typically by 16).

Compute shader dispatches are always defined in three dimensions, *x, y, z,* each with their own number of local workgroups (Figure 6). For our culling implementation, we only need to use one dimension, therefore we define our dispatch like so:

*CommandBuffer.Dispatch(instance_count / 16, 1, 1);*

The number of dispatched groups is calculated based on the total number of instances divided by the workgroup size to ensure complete coverage of all asteroid instances.

The invocation is organized into workgroups, each containing 16 threads as specified in the compute shader by the local workgroup size (*local_size_x* = 16). Therefore, the total number of workgroups dispatched is computed as *instance_count* / 16 to ensure all asteroid instances are processed. This division allows efficient GPU utilization, balancing workload evenly across multiple shader cores.

To dispatch and organize the compute shader within the engine, eight buffers and descriptors are initialized.

We have one uniform buffer to contain the camera's position, the origin of the asteroid field, and the frustum planes for culling computations. This buffer needs to be of uniform storage qualifier because its variables need to be updated each frame. Indeed, both the camera and the frustum planes need to move as the player flies around, and we need the asteroid field to be able to be displaceable too.

We have three read-only storage buffers that are loaded at the start of the scene: a buffer for the asteroid's positions that stores the world-space positions of each asteroid instance, another for their template type that stores identifiers for the type of asteroid template assigned to each instance (as we will see later, we have multiple different shapes of asteroids), and finally a buffer for their vertex index ranges that stores precomputed index ranges used to render different asteroid templates and LODs efficiently.

We have two write-only indirect draw buffers that hold indirect draw commands, which we will discuss in the next section.

Finally, we have an atomic counter buffer to track the number of visible instances, facilitating dynamic indexing into yet another write-only buffer that records visible instances.

## 3.3 Draw Indirect Data Structure

For our project, we made use of indirect draws to render our asteroids. An indirect draw command allows the GPU to execute rendering calls directly based on data stored within the GPU memory, without additional CPU overhead per draw call. This method reduces CPU-GPU synchronization, which significantly improves rendering efficiency, especially when rendering a large number of objects like in our case. We use Vulkan's *vkCmdDrawIndexedIndirect* to render our large number of instances directly on the GPU. For this, we had to prepare an indirect draw buffer containing draw parameters. In our compute shader, we have two indirect draw write-only buffers with the following structure of parameters: index count, instance count, first index, vertex offset, and first instance. The index count defines how many indices to read for rendering an object's geometry from the index buffer, and the instance count specifies how many instances of that geometry should be drawn. Setting that last parameter to zero skips rendering the geometry, which is important for our culling algorithm. The first index is the starting point within the index buffer from which indices are read. The vertex offset is added to each vertex index to efficiently reuse geometry data, and the first instance identifies the starting instance ID used by shaders to differentiate multiple instances during rendering.

## 3.4 Frustum Culling Algorithm

Here is a simplified version of our asteroid frustum culling algorithm in our compute shader. As mentioned previously, this algorithm runs in parallel for every asteroid instance.

**Algorithm 1** Asteroid Instance Visibility

```
$world\_position \leftarrow instance\_position + asteroid\_field\_origin$
$is\_visible \leftarrow true$

for each $frustum\_plane$ do
    if $dot(world\_position, frustum\_plane) + radius < 0$ then
        $is\_visible \leftarrow false$
        break
    end if
end for

if $is\_visible$ then
    $index \leftarrow atomicAdd(atomic\_counter, 1)$
    $visible\_indices[index] \leftarrow instance$

    $distance\_to\_camera \leftarrow length(world\_position - camera\_position)$
    $lod\_level \leftarrow clamp(distance\_to\_camera \times 0.05, 0, max\_LOD)$

    if $lod\_level > billboard\_threshold$ then
        $indirect\_billboard\_buffer[instance].instanceCount \leftarrow 1$
        $indirect\_asteroid\_buffer[instance].instanceCount \leftarrow 0$
    else
        $indirect\_billboard\_buffer[instance].instanceCount \leftarrow 0$
        $indirect\_asteroid\_buffer[instance].instanceCount \leftarrow 1$
    end if
else
    $indirect\_billboard\_buffer[instance].instanceCount \leftarrow 0$
    $indirect\_asteroid\_buffer[instance].instanceCount \leftarrow 0$
end if
```

We can separate the algorithm in three steps:

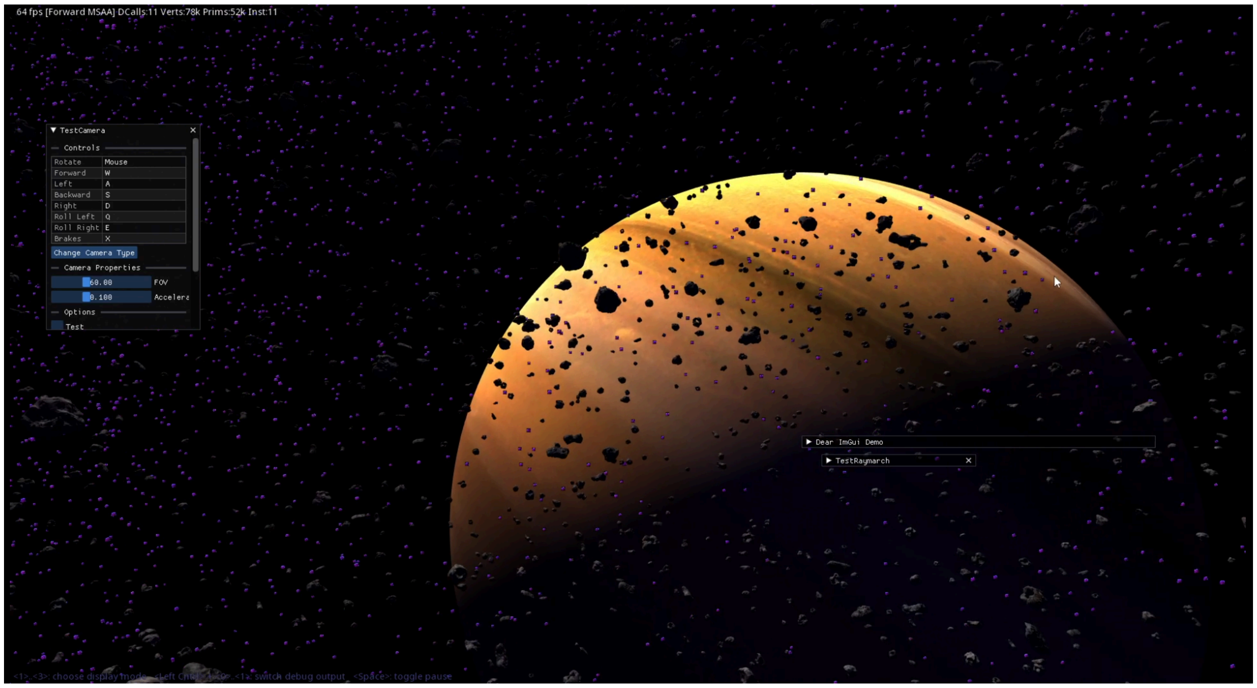
First, each asteroid instance position, stored in the position buffer, is transformed into world space by adding the asteroid field origin defined in our uniform buffer. The transformed positions are then individually checked against the view frustum using a function that iterates through the six frustum planes provided by our uniform buffer. For each asteroid instance, a dot product calculation checks its spatial relationship to each plane. After the result, we then add the radius of our roughly-spherical asteroid. If an instance is behind one frustum plane (indicating it is outside the camera's view), its instance count in their indirect draw buffer is set to zero to cull it from being drawn.

Then, an atomic counter buffer tracks the number of visible asteroid instances. Each time an asteroid passes the frustum checks, it increments the atomic count. An atomic counter is a special type of variable that allows safe incrementing or decrementing from multiple threads simultaneously on the GPU. It ensures that each thread sees a unique, consistent value without conflicts or race conditions. This counter's output is used to index into a visible index buffer, effectively compiling a compact list of indices corresponding exclusively to visible asteroid instances that is updated at each frame.

Finally, asteroids determined to be within the frustum have their mesh data assigned to their indirect draw buffer with an appropriate level of detail (LOD) based on their distance from the camera. The farthest ones are rendered as billboards, which we will see in more details in the next section.

## 3.5 Asteroid Templates and LOD Structure

We have five different template types for our asteroids, each with different shapes and sizes to give our asteroids more variety. Each of these templates are further subdivided into five different LODs, for a total of 25 different meshes. At the time of working on this project, the LODs and templates only varied in geometry, their textures and materials remained unchanged, although it would be a simple modification. We store all 25 meshes in a read-only buffer that organises these templates sequentially by indexing each template with their following LODs. For each mesh, we store its first index and index count so we can, for a given instance inside the frustum for the field of view, write them into the indirect draw buffer for rendering. The handling of vertex indices for each instance is done in the compute shader. It is not shown in **Algorithm 1** to not distract the reader from the essentials of the algorithm. The initial storage process is done before the asteroid scene loads and does not need to be redone again. An asteroid template is assigned to each instance randomly, while the LODs are defined by distance with a clamping function as seen in **Algorithm 1.** The shorter the distance, the lower the LOD number. An LOD level of zero indicates the highest level of detail possible, while a larger LOD value calls for lower details. Going from zero to four, we have a total of five different LOD meshes for each template. However, we defined our *max_LOD* count to be five, so our clamping function can return a larger LOD number than we have meshes for. This is because we reserve the very last LOD level for our billboards, which are stored in a different draw indirect buffer. Our billboards are our simplest meshes, with the lowest level of detail because they consist of only four vertices and a low-resolution texture. With GPU-based frustum culling and our LOD system with billboards, our engine is already capable of running scenes of more than one million asteroids at 60 FPS (frames per second) on a mid-range graphics card. Below in Figure 7 is a screenshot of the LOD system at work, with the billboards having bright placeholder texture in debug mode so we can differentiate them more easily.



```
64 fps [Forward MSAA] DCalls:11 Verts:78k Prims:52k Inst:11
```

| | |
|---|---|
| Rotate | Mouse |
| Forward | W |
| Left | A |
| Backward | S |
| Right | D |
| Roll Left | Q |
| Roll Right | E |
| Brakes | X |

```
60.00  FOV
0.100  Accelera
```

TestCamera
Controls
Change Camera Type
Camera Properties
Options
Test
Dear ImGui Demo
TestRaymarch
<1>..<3>: choose display mode  <Left Cntrl> + <0>..<1>: switch debug output  <Space>: toggle pause

**Figure 7:** Screenshot of the LOD system at work.

## 3.6 Orientating Billboards

At our lowest level of detail, we have billboards. Their geometry is implemented with a simple rectangle defined by four vertices forming a 2D square. For billboards to work properly, their plane of support needs to always be oriented towards the camera. We accomplish this by defining a special transformation in their vertex shader. For most applications, billboards have two ways of orienting themselves towards the camera, either by doing cylindrical. Or spherical billboarding [4]. Cylindrical billboards rotate only around a single axis, typically the Y-axis. This is especially useful for an object like a tree, where we want it to face the camera as we move horizontally around the object, but to remain upright when viewed from above or below. Indeed, allowing full rotation would make the object appear to tilt or uproot unnaturally, breaking the illusion that they are anchored to the ground. In our case, where we simulate asteroids in space, spherical billboarding is the better alignment. Spherical billboards rotate to face the camera from all view directions. We need this because asteroids in the void are not rooted to any direction and can be seen from any direction.

To orient a billboard such that it always faces the camera in the vertex shader, we use vectors pre-calculated by our view matrix and we apply the following calculations.

Let:

- $\mathrm{P}_{\text{world}}$ be the billboard's center position in world space.
- $\mathrm{R}_{\text{world}}$ be the camera's vector pointing to its right in world space.
- $\mathrm{U}_{\text{world}}$ be the camera's vector pointing above (up) itself in world space.

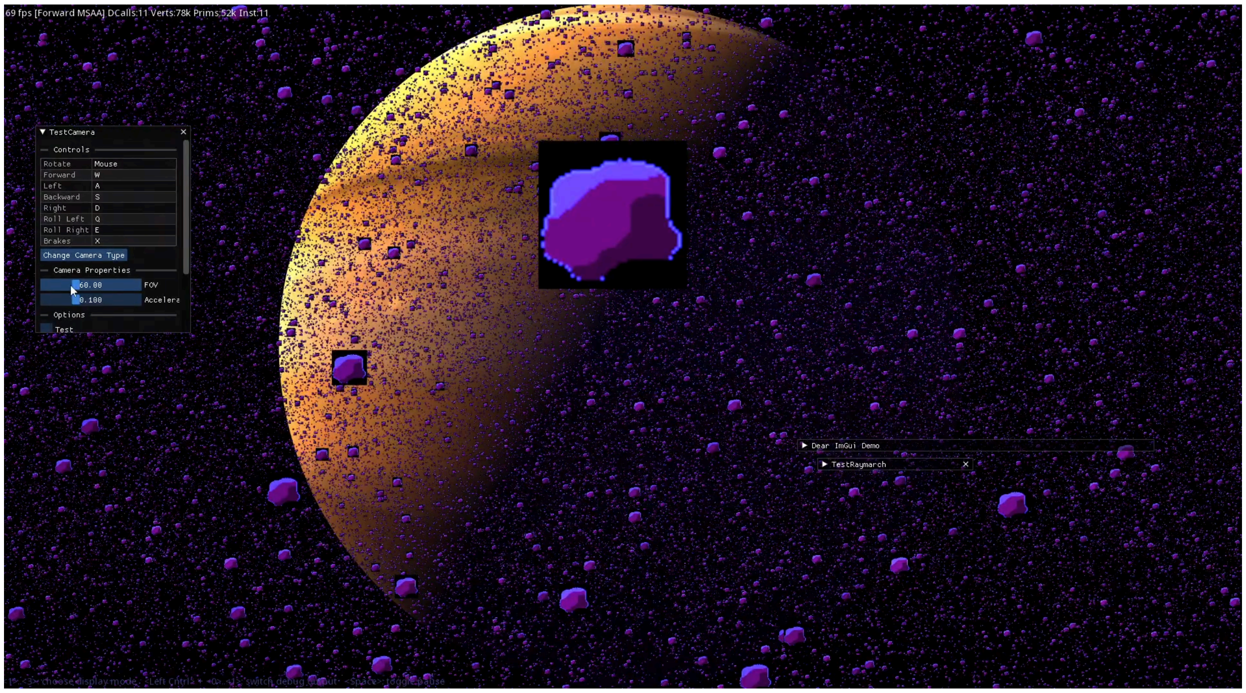
For each *x* and *y* coordinates in local space of a vertex belonging to the billboard rectangle, we compute our vertex to world space $\mathrm{V}_{\text{world}}$ :

$$\mathrm{V}_{\text{world}} = \mathrm{P}_{\text{world}} + x_{\text{local}} \cdot \mathrm{R}_{\text{world}} + y_{\text{local}} \cdot \mathrm{U}_{\text{world}}$$

Finally, we transform our world space vertex to clip space, $\mathrm{V}_{\text{clip}}$ , by multiplying it with our model-view-projection matrix ($\mathrm{M}_{\text{MVP}}$):

$$\mathrm{V}_{\text{clip}} = \mathrm{M}_{\text{MVP}} \cdot \mathrm{V}_{\text{world}}$$

The above method works because the camera's right and up vectors, extracted from the camera's orientation in world space, define the plane perpendicular to the camera's viewing direction. By offsetting each billboard vertex along these vectors, the quadrilateral is constructed to always lie parallel to the camera's image plane, regardless of the camera's position and orientation. This is illustrated in Figure 8.



```
69 fps [Forward MSAA] DCalls:11 Verts:78k Prims:52k Inst:11
```

```
▼ TestCamera
— Controls
Rotate      Mouse
Forward     W
Left        A
Backward    S
Right       D
Roll Left   Q
Roll Right  E
Brakes      X
Change Camera Type
— Camera Properties
60.00   FOV
0.100   Accelera
— Options
Test
```

**Figure 8:** Screenshot of billboards with placeholder textures aligning themselves to the camera.

Next, in the fragment shader we can apply the albedo and normal map textures. These are obtained by pre-rendering the "front view" of each asteroid template before the scene loads. We can then apply normal Phong shading to our billboard. The normal map will approximate what the asteroid would look like if it were lit in 3D. Our asteroids can seemingly transition from 2D to 3D, but obviously only if our camera does not change its orientation and if our asteroids remain still. Both of which are not true, because our camera can move and look in any direction, and the asteroids all rotate arbitrarily in space. So in our scenario, the billboards will immediately lose their synchronisation to our 3D asteroids. To solve this problem, we need to think about building a texture atlas with every asteroid orientation, depending on our camera's position and orientation relative to our billboards, and also depending on our asteroid's current transformation.

## 3.7 Billboard Texture Atlas

For our billboards to accurately simulate our 3D asteroids from far away, we need them to update their texture as the asteroid rotates or as the camera changes orientation. In many billboard applications in real-time engines, texture atlases are used to animate different angles of a 3D object onto a billboard. A common case is for foliage. A series of textures are kept in an atlas, where each texture is rendered from a different angle of the foliage all around their vertical axis. For that last example, the atlas would only need to extend to one dimension, because we only have to capture the foliage's yaw axis (y-axis). For our case, we need more than only the yaw axis. Before the scene loads, we rotate each asteroid shape at various angles in front of a camera that renders them into a texture. We then place each texture in a 2D texture atlas.

More precisely, we divide our billboard texture atlas into a grid, defining its resolution by $R_x$ columns horizontally (pitch/x rotations) and $R_y$ rows vertically (yaw/y rotations). For each cell of the grid located at coordinates $(i, j)$ we calculate pitch and yaw angles that evenly sample the asteroid's rotation space:

$$\theta_{pitch}(i) = \frac{2\pi \cdot i}{R_x} \qquad \theta_{yaw}(j) = \frac{2\pi \cdot j}{R_y}$$

This gives us evenly spaced rotations from 0 to 2π for each texture atlas tile. We then input these sampled orientations into a combination of rotation matrices to have a complete atlas of all pitch and yaw rotations of our asteroids, like the images (see Figure 9) shown for an atlas of 32 by 32 textures.

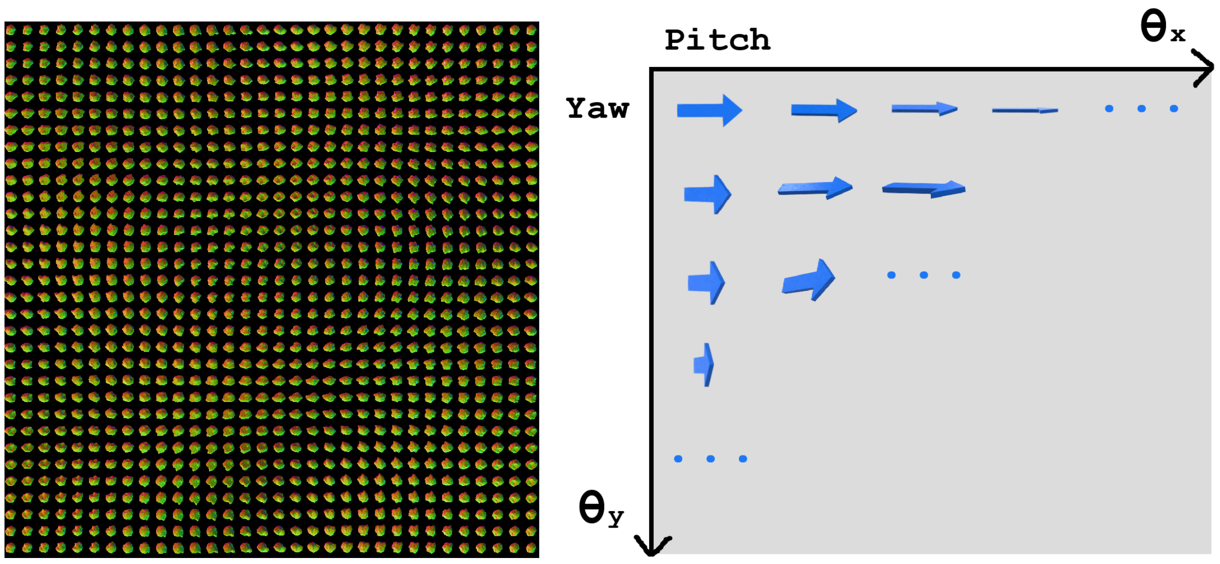


**Figure 9:** On the left, an asteroid's pitch and yaw texture atlas visualized in its normal map. On the right, a representation of its orientation with an arrow to help visualization.

However, we quickly fall into a major problem. We have pitch and yaw rotations, but we are missing roll. A 3D asteroid can rotate in all three axes of rotation, and the space engine camera can look at an object from any angle, Therefore, we would inevitably miss many possible asteroid orientations in our texture atlas. Suppose our camera faces an asteroid rotating purely on its yaw angle. If we do not move our camera, we would be able to accurately animate our asteroid by cycling through the first row of our pitch/yaw texture atlas. However, as soon as we move around the asteroid to observe it from a different angle, immediately we will face missing rotations, because from our point of view, the asteroid will now be doing a roll rotation that is absent from our atlas (see Figure 10).

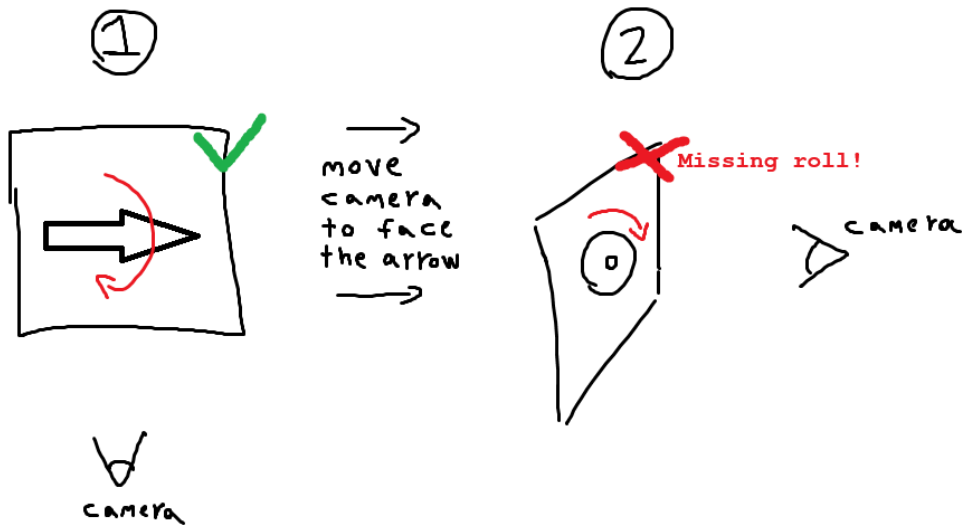


**Figure 10:** An arrow doing a pitch rotation from one view angle, and a roll from another angle.

The naive solution to this problem would be to build a 3D texture atlas that includes all rotation angles. But, because our texture atlas is 32 by 32 textures, increasing its dimension would make it 32 times larger! Considering we have five different asteroid shapes to load into our GPU, five 3D texture atlases of 32 by 32 by 32 textures (163,840 in total) would be an unacceptable demand in memory size.

The actual solution to this problem is, in fact, simple. We keep our 2D pitch and yaw texture atlas. However, when a roll is being performed, either by the camera or by the asteroid, we simply roll the billboard itself directly in its vertex shader to simulate it. Although the idea is simple, we are only halfway done. We still need to find the mathematical function (discussed in the next section) to map a 3D asteroid rotation to a 2D atlas, with a proper billboard roll.

In the end, for the billboard fragment shader, where we apply its textures, we load a texture atlas of five texture atlases for each asteroid shape, which in turn has 32 by 32 textures. In total, this corresponds to one large texture atlas of 160 by 32 textures (tiles) (see Figure 11). The tile selection is done in the billboard's vertex shader where we calculate its $(i, j)$ index given as input to the fragment shader.

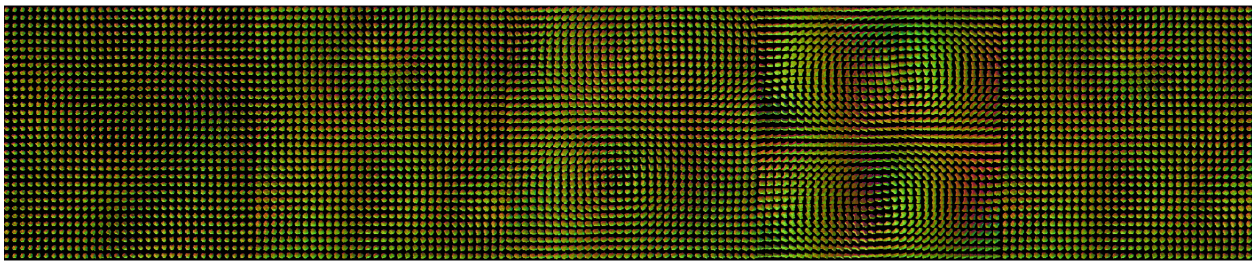


**Figure 11:** Texture atlas of texture atlases of asteroid rotations for five shape templates.

## 3.8 Animating Billboards

Every asteroid instance is given a random rotation axis in a 3D vector:

$$v = (r_x, r_y, r_z) \mid r_x, r_y, r_z \in [0, 1]$$

The vector is normalized and converted into a quaternion. To represent rotations using quaternions, we convert from the axis-angle representation to a quaternion form. Given an arbitrary vector $v$, we interpret its direction as the rotation axis and its magnitude $||v||$ as the rotation angle θ.

We thus define:

$$\theta = ||v|| \qquad u = \frac{v}{||v||}$$

A quaternion $q$ representing a rotation of angle θ about a unit axis $u$ is defined:

$$q = (\cos\frac{\theta}{2}, u\sin\frac{\theta}{2})$$

The half angle α is defined as:

$$\alpha = \frac{||v||}{2}$$

We can calculate each dimension of a quaternion by combining the previous equations:

$$q_x = v_x \cdot \frac{\sin(\alpha)}{||v||},\ q_y = v_y \cdot \frac{\sin(\alpha)}{||v||},\ q_z = v_z \cdot \frac{\sin(\alpha)}{||v||},\ q_w = \cos(\alpha)$$

We send the resulting quaternion to both our 3D asteroid vertex shader and to our billboard vertex shader as instanced attributes that are unique to each vertex group tied to their instance. As a 3D asteroid rotates depending on its given quaternion rotation, when the instance transforms into its billboard counterpart, the same quaternion remains. This is important so we can synchronise the 3D asteroid's rotation to the tile that we select in its texture atlas.

The texture tile selection for the atlas depends on knowing the asteroid's relative rotation to the camera. Indeed, we need to take into account both the 3D asteroid's rotation and the camera's orientation in space to deduce which tile we need to display on the billboard. After converting the asteroid's quaternion rotation to a proper transformation matrix, the relative rotation is found by multiplying the inverse of the 3D asteroid's rotation matrix by the camera's rotation matrix. Let $M_C$ be the camera's rotation matrix, extracted from its view matrix, and $M_A$ be the asteroid's rotation matrix, then the relative rotation $M_R$ is found like:

$$M_R = M_C \cdot M_A{}^T$$

Note here we are using the transpose of the asteroid's matrix instead of its inverse to save on computation, because a rotation matrix is orthogonal and so its inverse is equal to its transpose.

Now, for us to find the proper index to select in our texture atlas, we need to solve for *i* and *j* in our texture atlas equation in Section 3.7.

$$i = \frac{\theta_x \cdot R_x}{2\pi} \qquad j = \frac{\theta_y \cdot R_y}{2\pi}$$

Pitch $\theta_x$ and yaw $\theta_y$ are extracted from the relative rotation matrix ($M_R$) by converting it to an Euler vector. Finally, we round up the result to an integer before sending the (*i, j*) results to the fragment shader, along with its template type number, so it can select the correct tile in the texture atlas for display.

We have pitch and yaw, but we are still missing the roll rotation. As we discussed in the previous section, we do not have roll orientations in our texture atlas. Our solution to simulate asteroid rolls is to roll the billboards themselves. To do so, we first calculate the direction vector from the billboard to the camera. This is so we can rotate the asteroid along this axis, facing the camera. Let $P_{\mathrm{cam}}$ be the camera's position in world space and $P_{\mathrm{bb}}$ be the billboard's instanced position in world space. We subtract them to get the direction axis $V_{\mathrm{dir}}$:

$$V_{dir} = ||P_{cam} - P_{bb}||$$

Then, we simply multiply this axis by the roll angle $\theta_z$ extracted from the Euler vector we got from the asteroid's rotation matrix and multiply it to the camera-billboard axis. After converting the result to a matrix, we get a roll rotation $R_z$:

$$R_{\mathrm{z}} = \mathrm{EulerToMatrix}\left(V_{\mathrm{dir}} \cdot \begin{bmatrix} 1 \\ 1 \\ \theta_z \end{bmatrix}\right)$$

We can then use this rotation matrix to transform our billboard matrix *B* into *B*'

$$B' = R_z \cdot B'$$

And finally, we can use the billboard's *x* and *y* coordinates like in Section 3.6 to align it to the camera, without affecting its roll.

With the billboard's roll and the texture atlas pitch and yaw selections, we now cover all of the 3D asteroid's rotations. Running the previous equations in real time, we achieve a very accurate simulation of the 3D asteroids with our billboards (see Figure 12 and Figure 13).

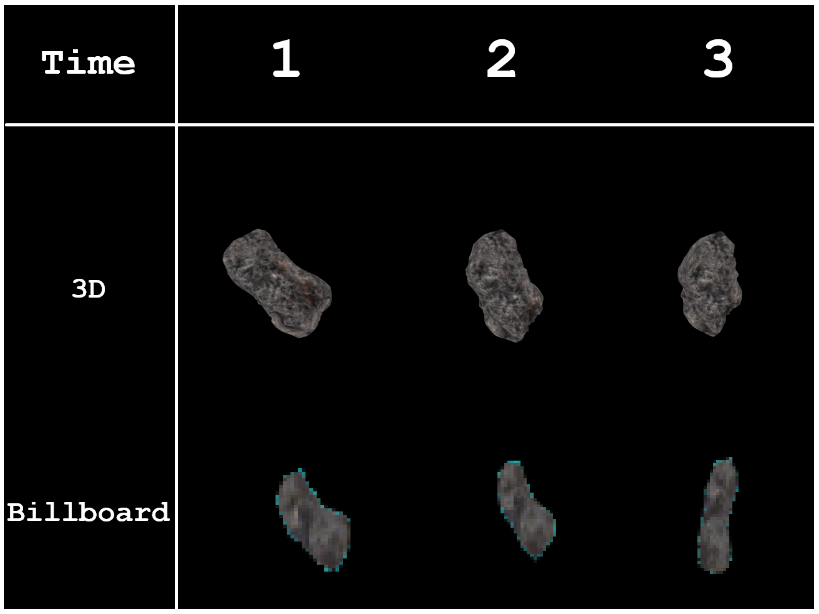
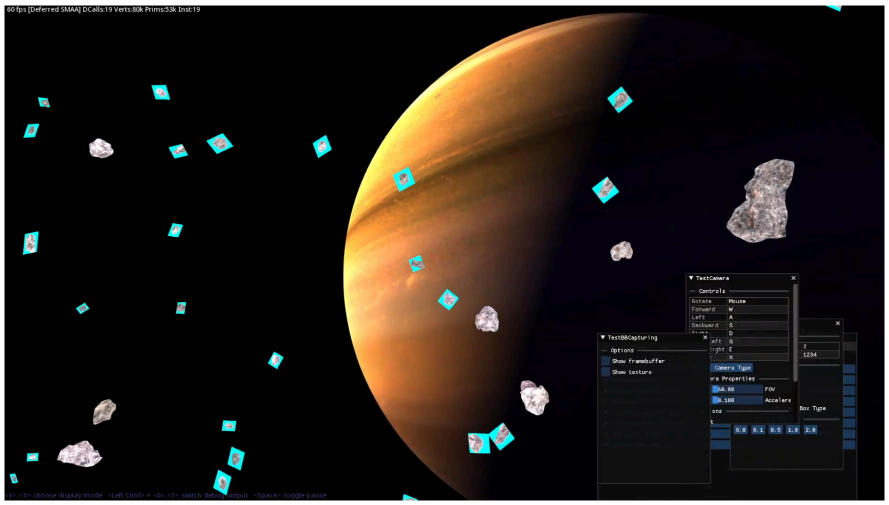


**Figure 12:** A 3D asteroid rotating as its billboard follows the same transformations.



```
60 fps [Deferred SMAA] DCalls:19 Verts:80k Prims:53k Inst:19
```

**Figure 13:** Screenshot of [billboards in debug mode to visualize their texture atlas selections.]

Figure 13 shows the asteroids and billboards being fully lit, as well as the billboard background being opaque cyan, for us to be able to tell the difference when changing LODs. In practice, the billboards appear farther away and their background is transparent, so the final result makes them barely noticeable as we will see in the next chapter.

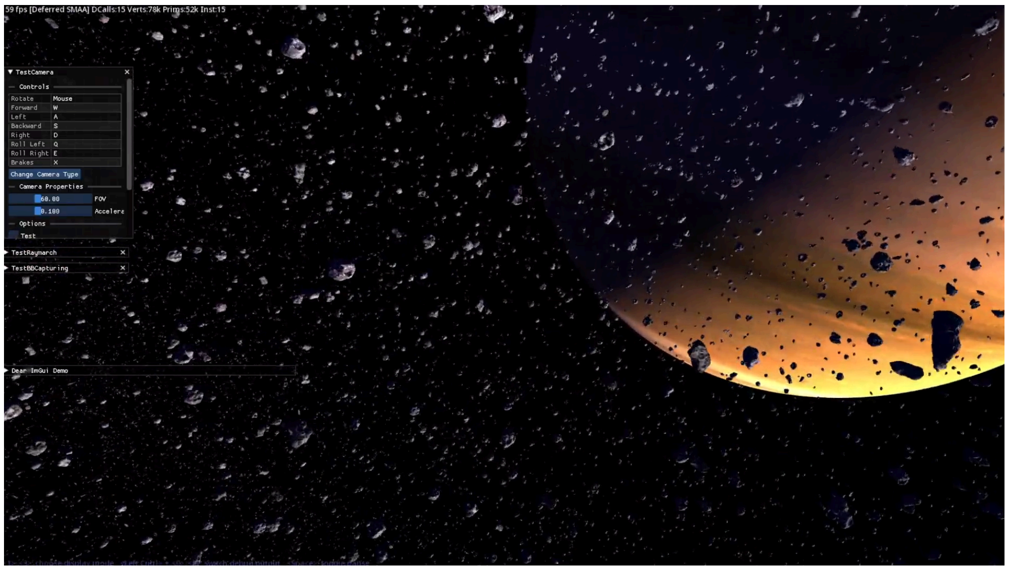
# CHAPTER 4

# RESULTS

In this chapter, we discuss the final results of the project completed in this internship. We look at performance metrics such as frames per second and GPU memory usage. Then, we compare our work to similar projects.

## 4.1 Performance

We ran our performance tests on an NVIDIA RTX$^{TM}$ A2000 with 6GB of VRAM and an Intel Core$^{TM}$ i7-9700 at a resolution of 1920 by 1080 pixels. From the data we gathered from *Steam analytics*, these specifications are representative of a typical midrange desktop configuration for the average *SpaceEngine* user. We ran our tests on a million asteroid instances with five 3D LOD levels going from ~1400 triangles down to ~500 triangles, and with billboards of only two triangles and four vertices.



```
59 fps [Deferred SMAA] DCalls:15 Verts:78k Prims:52k Inst:15
```

| | |
|---|---|
| Rotate | Mouse |
| Forward | W |
| Left | A |
| Backward | S |
| Right | D |
| Roll Left | Q |
| Roll Right | E |
| Brakes | X |

| | |
|---|---|
| 60.00 | FOV |
| 0.100 | Accelera |

**Figure 14:** Screenshot of an asteroid field with a million instances running at 60 FPS.

With this set up, we are successfully able to render our scene at 60 frames per second (FPS) at the worst case, that is the asteroid field completely on screen as zoomed out as much as possible while still having all LODs shown. Because we are using large textures regardless of LODs

(4096 by 4096 diffuse and normal map textures for each template), we are consuming an unnecessarily large amount of memory at around 1.1 GB of VRAM (GPU memory). Although this could be easily remediated by using lower resolution textures, especially since the majority of the asteroids are displayed as billboards or at low level of detail. This scene is performance-wise GPU-bound because of our heavy use of GPU-based calculations while our CPU workload remains minimal.

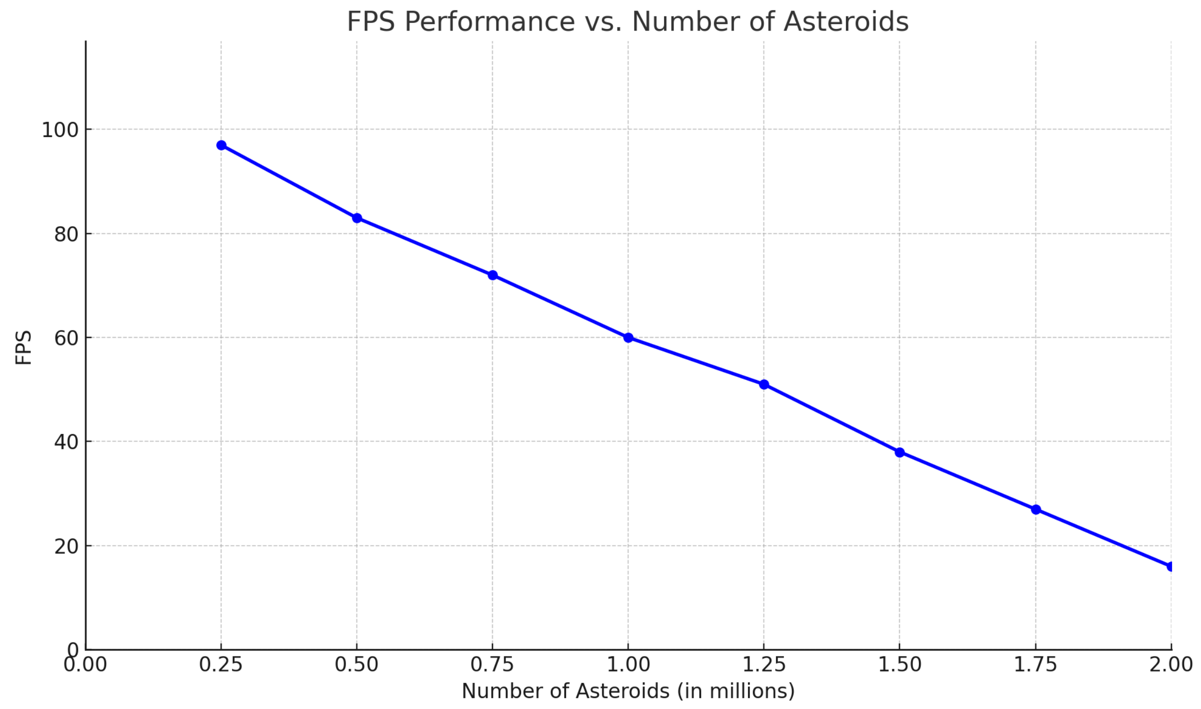


**Figure 15:** Performance depending on the number of asteroid instances in a constant volume.

The chart above illustrates the relationship between FPS and the number of asteroid instances, tested on a computer with the specifications mentioned previously, within unchanging scene bounds (otherwise if the scene size could change we could maintain high FPS at will because of our frustum culling). At 250,000 instances, we have 97 FPS, decreasing to 83 FPS at 500,000 instances and 72 FPS at 750,000. Our target of 60 FPS is maintained at one million asteroids. Beyond this, FPS starts dropping: 51 FPS at 1.25 million, 38 FPS at 1.5 million, 27 FPS at 1.75 million, and 16 FPS at two million instances. We can observe from the chart that rendering efficiency decreases almost linearly with the studied scene complexity. Although, without changing the volume where asteroids appear, beyond 1.5 million asteroids the scene starts to look cramped and unrealistic. Billboards are the main reason for maintaining performance as the number of asteroids increases, because they take up the vast majority of asteroids rendered, depending on the camera's positioning. Without them, we have around 25 FPS at a million asteroids, on average. And without GPU frustum culling (using CPU frustum culling instead, with mesh LODs) we have 20 FPS at only 100,000 asteroids. The software can not run in real time at one million without the above two techniques mentioned. Lighting shading takes an insignificant amount of computation for our scene. Removing our Phong shading for asteroids and billboards merely increases our performance by one FPS, at best. The main way we have to manipulate performance at the expense of visual accuracy is to simply change the distance at which billboards and low-resolution meshes appear in our LOD system. In the future this will most likely be a performance option for users to manipulate at their discretion.

Visual-wise, the main aesthetical concern with our billboards is their lack of 3D geometry with regards to shading. Although our normals can simulate a lot of the intended shading we want on them, we can not simulate the 3D depth of their "real" counterparts. On a resolution of 1920 by 1080 pixels, they are harder to notice but on higher resolutions they might be more noticeable as they pop up on screen. To dissimulate the effect we would need to implement some sort of fade-in and fade-out effect in-between LODs.

## 4.2 Comparison

The majority of video games do not attempt to render such a large quantity of instances in their scenes. From observation, it would seem that most space-based video games use different methods to render asteroid fields. In the game *Elite Dangerous* by *Frontier Developments*, compared to our implementation, fewer asteroids are rendered on screen. To maintain performance, the game only renders asteroids close enough to the camera and culls out those too far away. To hide asteroids disappearing suddenly from view, they blend with distance fog to visually mask distant asteroids. Although not scientifically realistic, the implemented fog is perceptually convincing and makes for a visually pleasing scene. In contrast, *Starfield* by *Bethesda Game Studios* appears to render even fewer asteroids than *Elite Dangerous* or our implementation, but with significantly higher visual fidelity. The asteroids in *Starfield* have comparatively very detailed geometry and texture resolution. *Starfield's* approach favors close-up visual detail and cinematic presentation, and likely makes use of numerous LOD transitions instead of encumbering itself with the overhead of rendering a large number of instances.

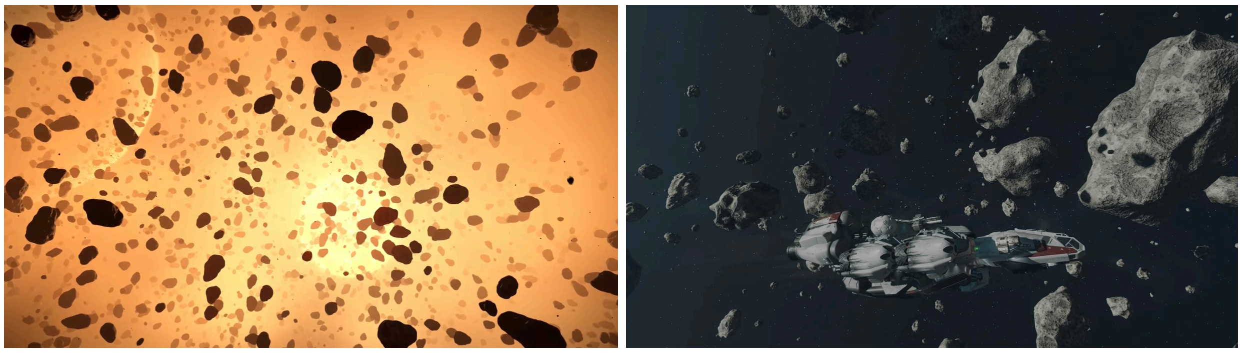


**Figure 16:** Screenshot of an asteroid field in *Elite Dangerous* (left) and in *Starfield* (right).

# CHAPTER 5

# CONCLUSION

In this report, I presented the main project I worked on while at *Cosmographic Software LLC.* I applied GPU-based graphics optimization techniques within *SpaceEngine*, a real-time astronomical visualization software. The primary objective, rendering millions of asteroid instances efficiently at a stable performance of 60 frames per second on a mid-range hardware, was achieved through the implementation of GPU-based frustum culling, dynamic level of detail, and a special animated billboarding solution.

The developed GPU-based frustum culling system utilizes Vulkan's compute shaders to significantly reduce unnecessary CPU processing by culling asteroid instances outside the camera's view, using the GPU’s parallel processing. The implementation of a dynamic LOD system further optimized rendering performance by adjusting mesh complexity based on camera distance. The billboarding approach I implemented, with a texture atlas to animate asteroids, effectively simulated the rotation of 3D asteroids without compromising visual realism. However, such texture-based asteroids would not cast shadows on the surface of the planet, which may affect realism.

The results of this internship include not only substantial performance improvements for asteroid rendering but also laid the foundation for future applications within the engine, such as debris rendering on the surface of planets. Debris would use the same suite of techniques used for the asteroids, except a 2D atlas of textures to animate their billboards would not be needed, since debris are not supposed to be able to rotate arbitrarily in any direction.

There are many areas to improve, some we mentioned in Chapter 4 on results. These include adding fading between LODs to make transitions less noticeable, and adding performance control options for the user such as LODs distance selection. Other future work would be extending the current billboarding system to also support billboard clouds [5]. It is a useful approach to render impostors using multiple billboards that support viewing an object from any angle without using a very large texture atlas of multiple rotation views. Another area to think about is how our billboards will work with Virtual Reality. Indeed, *SpaceEngine* supports Virtual Reality in OpenGL, while its Vulkan implementation is still a work in progress. Once it is implemented, it is important to test our project in Virtual Reality, to evaluate how billboard alignment works with two views (human eyes) and to validate if our tile selection functions to simulate asteroid rotation depending on camera position still work.

Another direction for future optimization would be to explore LOD systems for groups or hierarchies of asteroids, instead of treating each asteroid instance individually. Group-level culling and LODs could significantly reduce GPU workload when asteroids are distant or clustered.

Outside of specifically asteroids, one thing that will improve realism for asteroid fields and asteroid rings around planets is the simulation of dust-like particles. In a real universe, after asteroids collide with each other, it would make sense to have much smaller rock particles floating around. This would most likely require a particle system to be implemented, as billboards would most likely not be suited for the task. An extension to this would be the use of participating media to simulate volumetric effects such as light shafts in dusty regions, which would probably make for good visuals. Lighting effects could also be improved by simulating the glow of light around asteroids, when facing a light source such as a star. Adding a glow effect, either through general post-processing or with a specialized asteroid fragment shader, could help simulate how real light would interact from stars to asteroid surfaces.

Most of the *SpaceEngine'*s features such as stars, navigation physics, and other astronomical objects are written in OpenGL. We are still in the progress of implementing them in Vulkan. As more features get implemented in Vulkan, we will need to see how they interact with our asteroid field performance-wise, and adjust accordingly. One thing we know will be a problem are nebulas. They are implemented as volumetric clouds, so they take a lot of resources to run. Asteroid fields might not be able to render inside them, so we will need to either find a different way to display them inside nebula clouds or to simply disallow to render both at the same time.

In the end, the work done in this internship has provided valuable hands-on experience in rendering programming and deepened my expertise in the Vulkan API. More specifically I learned a lot about compute shader programming, real-time pipeline synchronisation (between shaders and CPU-GPU communication), and graphics optimization techniques. All of this has given me a strong foundation for further learning in the domain of real-time Computer Graphics.

# REFERENCES

[1] Hipparcos Catalog https://www.cosmos.esa.int/web/hipparcos

[2] New General Catalogue of Nebulae and Clusters of Stars https://ngcicproject.observers.org/

[3] Hugues Hoppe, "Progressive Meshes" hhoppe https://hhoppe.com/proj/pm/

[4] Tamas Umenhoffer, Laszlo Szirmay-Kalos, Gabor Szijarto "Spherical billboards and their application to rendering explosions" ResearchGate https://www.researchgate.net/publication/200019094_Spherical_billboards_and_their_application_to_rendering_explosions

[5] X. Decoret, F. Durand, F.Sillion, , J. Dorsey "Billboard Clouds for Extreme Model Simplification"  Yale University https://graphics.cs.yale.edu/publications/billboard-clouds-extreme-model-simplification

[Figure 1] Mykhailo Moroz,"Visualizing General Relativity" Space Engine https://spaceengine.org/articles/visualizing-general-relativity/

[Figure 2] Mohd Shahrizal, Abdullah Mohd,Tengku Sembok, "Effective Range Detection Approach for Ancient Malacca Virtual Walkthrough" ResearchGate https://www.researchgate.net/publication/238687498_Effective_Range_Detection_Approach_for_Ancient_Malacca_Virtual_Walkthrough

[Figure 3] Saint Thomas, "LOD Document" Game Development Diary https://moderndynamics.wordpress.com/year-2/object-orientated-design/lod-document/

[Figure 4] Simone Barbieri, Ben Cawthorne, Zhidong Xiao, Xiaosong Yang "Repurpose 2D Character Animations for a VR Environment Using BDH Shape Interpolation", ResearchGate https://www.researchgate.net/figure/This-example-shows-how-the-billboarding-works-B1-B2-and-B3-are-the-billboards-which_fig2_320730778

[Figure 5] Sasha Willems, "Compute Culling Screenshot" Vulkan Examples https://github.com/SaschaWillems/Vulkan/blob/master/

[Figure 6] Jonas Sorgenfrei, "Introduction to GPU Computing" learnopengl https://learnopengl.com/Guest-Articles/2022/Compute-Shaders/Introduction

[Figure 16 (left)] Rosy, "Elite Dangerous: Odyssey - Beautiful Planetary Ring" youtube https://www.youtube.com/watch?v=JMnsunfz0Hk

[Figure 16 (right)] Supernaut Games, "Starfield - The Tranquility of an Asteroid Field - 4K HDR Showcase Series X" youtube https://www.youtube.com/watch?v=O1G4j088fGQ