UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de la Imatge i la Tecnologia Multimèdia

Optimization Techniques for a 2D Engine

Final Degree Project

Video Games Design and Development Degree

David Tello Panea

2021-2022

Director: Jesús Díaz García

Index

Summary

This document contains the description of the development of a small 2D engine written in C++. The focus of this project is to implement various optimization techniques to have a performant application.

The result is a small 2D engine that can be executed in any Windows machine and can handle more than 10.000 entities interacting with each other in real time.

The source code for this project is public and under the MIT License and can be found in the Github repository in the following link: https://github.com/DavidTello1/2D-Renderer

Key Words

Optimization, 2D Engine, Object-Oriented, Data-Oriented, Batch Rendering, Space Partitioning.

Tables Index

Figures Index

Glossary

- Efficiency[(W1)]: refers to the amount of work that a program needs to do, it is governed by the algorithm used. So the less work needed, the more efficient the algorithm is, which results in a faster program.

- Performance[(W1)]: refers to how fast a program can do the work needed, it is governed by the data structure. If you have the data structured in a way that the hardware can access it very quickly, the program will be more performant.

- Renderer: it is a program used for rendering. Rendering or image synthesis is the process of generating an image from a 2D or 3D model by means of a computer program. The resulting image is referred to as the render.

- Game Engine: it is a software framework primarily designed for the development of video games, and generally includes relevant libraries and support programs. The "engine" terminology is similar to the term "software engine" used in the software industry.

- GPU[(W2)]: stands for Graphics Processing Unit, it is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device. Modern GPUs are very efficient at manipulating computer graphics and image processing;

- CPU[(W2)]: stands for Central Processing Unit, it is also called a central processor, main processor or just processor. It is the electronic circuitry that executes instructions comprising a computer program. The CPU performs basic arithmetic, logic, controlling, and input/output (I/O) operations specified by the instructions in the program.

- Cache[(w3)]: it is a hardware or software component that stores data so that future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation or a copy of data stored elsewhere.
A "cache hit" occurs when the requested data can be found in a cache,

while a “cache miss” occurs when it cannot. Cache hits are served by reading data from the cache, which is faster than recomputing a result or reading from a slower data store; so, the more requests that can be served from the cache, the faster the system performs.

1. Introduction

This document analyzes the creation and development of a 2D Rendering Engine focusing on performance and optimization. A 2D Rendering Engine is an application that renders scenes or helps to create small games. It is 2D because this project will only focus on 2D scenes and objects.

Starting by creating a basic application, various optimizations are implemented one at a time until the final result is achieved. By implementing optimizations the aim is to achieve a performant application that is able to process the interactions of lots of objects in real time.

Some of the issues an engine can have are the bad structure of data that results in lots of cache misses, too many calls to a graphics card's driver where each call has a significant time cost or algorithmic decisions that lead to comparisons with O(N^2) costs.

The solutions we are going to implement in this project are structuring the data in a data-oriented design to avoid cache misses, batching render calls to reduce calls to the GPU and a space partitioning technique to help with comparative operations.

1.1 Motivation

The main motivation for this project is to develop a small application that can be later expanded to create a game engine, it can help other developers to just focus on other parts of engine programming and build upon this base.

During the degree I learnt about game engines and wanted to dive deeper into the subject, so this was the perfect opportunity to learn new things and test my skills.

By developing a small application instead of making a big and complex one, the focus can be shifted to optimization and performance. We can then analyze the results and reason about the improvements of a specific optimization we made.

1.2 Problem Statement

The main problem this project is trying to solve is the cost and low performance of executing code from an engine that has not taken into account various optimization factors.

This project focuses mainly on performance and efficiency, these are topics that have only been mentioned during the degree so it is something new for me so I have to expand my knowledge in this area in order to develop the project.

Efficiency and performance are two terms that can be easily confused with each other, to keep things clear we will briefly explain what they are.

Efficiency refers to the amount of work a program needs to do and is governed by the algorithm used, so that the better the algorithm the more efficient the program will be.

Performance refers to how fast you can do this work, it is governed by the hardware used and how the data is structured. The faster the memory-access the more performant the program will be.

Efficiency and performance are not dependent on one another. You can be efficient without being performant if you do very little work to get the job done but do it slowly, and you can also be performant without being efficient if you do a lot of work to get the job done but do it very fast.

What we are aiming for is to be both efficient and performant.

So, basically, the problem to solve are the factors that make an application or engine not as optimal as it could be, factors that include the performance and efficiency of it.

These problems have already been solved before, so this project isn't trying to solve something new but to gather these solutions and implement them in a small application that is efficient and performant.

By creating a small demo and documenting the process, other developers can use this project to expand it or use it as a guide for their own projects.

In conclusion, this is the problem we are going to solve. Next we will see the objectives set for this final degree project.

1.3 General Objectives

The main objective of this project is to develop a 2D Rendering Engine from scratch and implement various optimization techniques that will improve the efficiency and performance of the application.

Every time a new optimization is implemented, we will analyze the result and compare it with previous iterations to ensure there has been an improvement.

The final result should be able to handle 10.000 objects in the scene at a minimum of 60 FPS, every object in the scene should be able to interact with each other and, to show the results, a small application with some degree of interactivity will be created as a demo.

In summary, the general objectives are:

- 2D rendering from scratch.
- Applying optimization techniques that improve the performance to achieve the desired result.
- Documenting the process and creating a demo to make the project accessible and didactic.

1.4 Specific Objectives

To be able to achieve all the general objectives stated above, we need to break each objective into more specific objectives.

1.4.1 Framework

- **Basic Application**: create a basic application to develop the engine, structure the code in a modular way and keep functionalities separated to help with organization.

- **Resource Management:** be able to load textures, shaders and other resources into the engine and manage them accordingly.

1.4.2 Rendering

- **OpenGL:** implement OpenGL library and allow 2D rendering in the application.
- **Batch Rendering:** putting objects into batches to reduce calls to the GPU and allowing many objects to be drawn at once. It improves the performance of the rendering pipeline.
- **Clipping:** only drawing the objects inside the camera view frustum. It improves the performance of the application and makes it more efficient by discarding unnecessary draw calls to the GPU.

1.4.3 Scene Management

- **Entity Component System:** entities are only an index, components hold the data in arrays and systems operate on the data. It focuses on improving the efficiency of the application by keeping data tightly packed and makes it more performant when accessing it.

1.4.4 Physics

- **Rigidbodies:** physics apply to objects and they interact with each other in a realistic manner.
- **Colliders:** objects can collide with each other and the physics system will react to it accordingly.
- **Space Partitioning:** space is divided into sub-spaces. Physics operations only check for objects in the same space division, which results in more efficiency as there are less comparisons, and it also improves the performance of the physics system.

1.4.5 User Interface

- **Render Stats:** users can easily see visually the most important stats of the application.
- **Settings:** users can easily modify some parameters of the scene.

- **Adding and Removing Objects:** users can add and remove objects from the scene.

1.4.6 Demo

- **Debug Mode:** there is an option to see the debug features of the application to better see the inner functioning of the demo.
- **Build:** have an interactive demo that showcases the results of the project.

1.4.7 Documentation

- **Step by step:** each version of the application has its own dedicated part to explain the concepts and optimizations implemented.
- **Profiling and comparisons:** after each implementation, tests are done and compared with previous versions.
- **Graphs:** to better show the performance of each version there are some graphs showing the improvements of every implementation.

1.5 Project Scope

In this part we will explain the scope of what the project tries to achieve and who would benefit from it.

There is not a specific target as this is just a project to learn new skills. But it can be used as a base for more complex applications like a 2D Engine or a graphics renderer.

The beneficiaries of the project would be people who want to develop a more complex application using this project as a base or just want to have a working example to guide them when making their own application.

The scope has been limited to just the rendering engine with some basic physics to have interaction between objects, it should have all the basic functionalities to create a small game and be performant even if there is a large number of entities.

We could have Vulkan or other graphics libraries but that would be too much for the scope of the project, for this reason OpenGL was used as I had previous knowledge about it.

2. State of the Art

There are already many applications that do the same tasks this project addresses. In this part we will analyze the current solutions and the applications that exist in the market giving some details about the optimization techniques used.

2.1 Engines

There are many game engines but the three most popular ones nowadays are Unity, Unreal Engine and Godot. We will now analyze each one of them and the opportunities they give for developing optimized executables.

2.1.1 Unity[W4]

Unity is a cross-platform game engine developed by Unity Technologies, first announced and released in June 2005 at Apple Inc.'s Worldwide Developers Conference as a Mac OS X-exclusive game engine. The engine has since been gradually extended to support a variety of platforms.

It is considered easy to use for beginner developers and is popular for indie game development.

The engine can be used to create 3D and 2D games, as well as interactive simulations and other experiences.

One of the most relevant optimizations for the project that this engine uses is Unity's Data-Oriented Tech Stack (DOTS)) which is a combination of technologies that work together to deliver a data-oriented approach to coding in Unity. This allows users to build projects that are better suited to their target hardware and are therefore more performant.

The three main pillars of DOTS are:

- The Entity Component System (ECS), which provides the framework for coding using a Data-Oriented approach.
- The C# Job System, which provides a simple method of generating multithreaded code.

- The Burst Compiler, which generates fast and optimized native code.

For the physics calculations, Unity uses a 2D rigid body simulation library for games called Box2D. This library allows physics simulation with a variety of shapes and constraints.

For rendering, Unity supports Vulkan, Metal, DirectX and OpenGL APIs. As per optimization techniques used we can find GPU instancing, static and dynamic batching, culling and many more.

In the following figure we can see a screenshot of Unity's Tiny Racing demo, powered by DOTS.

F 2.1.1 - Unity DOTS

2.1.2 Unreal Engine[W5]

Unreal Engine is a game engine developed by Epic Games, first showcased in the 1998 first-person shooter game *Unreal.* Initially developed for PC first-person shooters, it has since been used in a variety of genres of 3D games and has seen adoption by other industries, most notably the film and television industry.

Written in C++, Unreal Engine features a high degree of portability, supporting a wide range of platforms.

The latest version is Unreal Engine 5, which was released in early access on May 26th, 2021.

About optimizations, the main ones related to the project are:

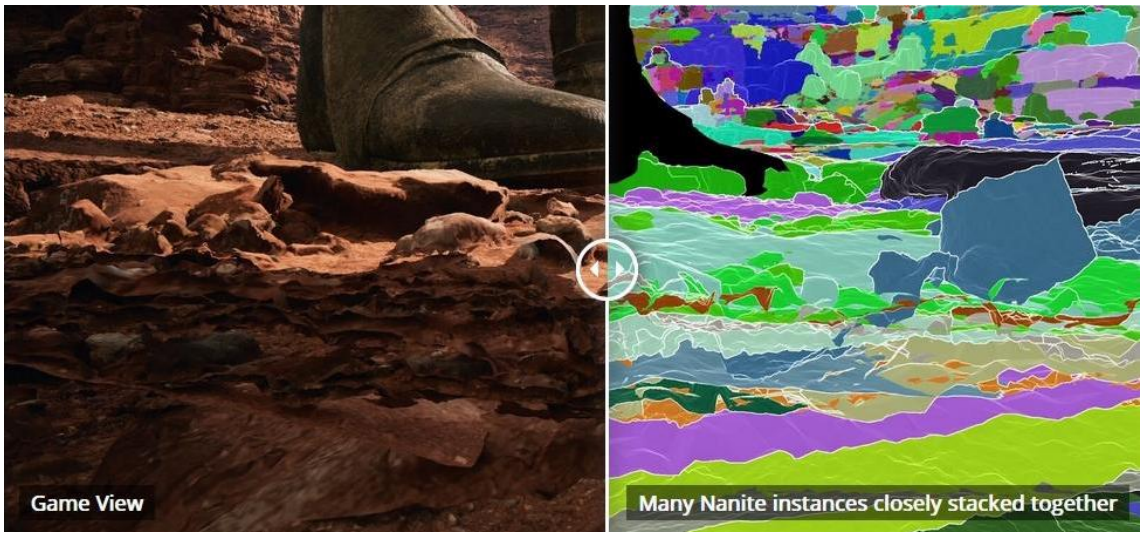
For the physics simulation, Unreal Engine uses PhysX, an open-source real-time physics engine middleware SDK developed by Nvidia as a part of Nvidia GameWorks software suite.

For rendering, it supports DirectX and Vulkan and offers a wide variety of options for shaders and post-processing.

Two big optimizations implemented in the latest version are Nanite and World Partition. Nanite is a virtualized micropolygon geometry system that allows the use of massive amounts of geometric detail while maintaining a real-time frame rate, and without any noticeable loss of fidelity. It streams and processes only the detail that can be perceived, largely removing poly count and draw call constraints.

World Partition is a system that automatically divides the world into a grid and streams the necessary cells. It helps to make the creation of open worlds faster, easier, and more collaborative for teams of all sizes.

In the following figure we can see an example of how Nanite works internally and the result that will be seen in-game.



F 2.1.2 - Unreal Engine's Nanite

2.1.3 Godot[W6]

Godot is a 2D and 3D cross-platform, free and open-source game engine.

It was initially developed by OKAM Studios from 2001. In February 2014 it was released to the public under the MIT License.

The development environment runs on multiple operating systems including Linux, macOS and Windows. Godot can create games targeting PC, mobile and web platforms.

Godot implements many optimizations but the main ones related to the project are the use of 2D batching, culling and the reuse of shaders and materials to decrease the number of draw calls which can be prohibitively high when treating each item individually. In addition, this means state changes, material and texture changes can be kept to a minimum.

For physics, Godot uses a proprietary engine called Godot Physics. This engine is almost entirely based on the Separating Axis Theorem algorithm for collision detection, which requires mathematically defining some axes used to check if a pair of objects are separated from each other.

For rendering, the OpenGL library is used.

In the following figure we can see Godot's interface and its animation system.

F 2.1.3 - Godot Engine's Animation System

2.2 Performance and Efficiency Issues

In software development there are many common pitfalls, in this part we will explain some of the most important ones regarding this project and how they can be solved.

2.2.1 Memory Architectures and Cache Misses[W25]

When creating a software application, memory architectures are an important topic to consider. Nowadays x64 is the usual mode as it supports vastly larger amounts of virtual memory and physical memory than was possible on its 32-bit predecessors, allowing programs to store larger amounts of data in memory.

CPUs don't load data directly from the system RAM, this is because system RAM is very slow to access. Instead, CPUs load data from a smaller, faster bank of memory called cache. Loading data from cache is considerably faster, but every time you try to load a memory address that is not stored in cache, the cache must make a trip to main memory and slowly load in some data. This delay can result in the CPU sitting around idle for a long time, and is referred to as a "cache miss". This time the CPU is waiting for data to be loaded implies a reduced performance of the program.

If you have an algorithm that loads small bits of data from randomly spread out areas of main memory, this can result in a lot of cache misses and, a lot of the time, the CPU will be waiting around for data instead of doing any work. If we make the data accesses localized, or even better, access memory in a linear fashion like a continuous list, then the cache will work optimally and the CPU will be able to work as fast as possible.

A solution to this problem is Data-Oriented Design (DOD), a program optimization approach motivated by efficient usage of the CPU cache and mainly used in video game development. The approach is to focus on the data layout, separating and sorting fields according to when they are needed, and to think about transformations of data. The parallel array or structure of arrays (SoA) is the main example of data-oriented design. It is contrasted with the array of structures (AoS) typical of object-oriented designs.

2.2.2 GPU Communication Bottlenecks

When drawing many objects in a scene by performing a draw call for each one of them, you'll quickly reach a performance bottleneck because of the many draw calls. Compared to rendering the actual vertices, telling the GPU to render your vertex data eats up quite some performance since the graphics library must make necessary preparations before it can draw the vertex data (like telling the GPU which buffer to read data from, where to find vertex attributes and all this over the relatively slow CPU to GPU bus). So even though rendering the vertices is super fast, giving the GPU the commands to render them isn't.

It would be much more convenient if we could send data over to the GPU once, and then tell the graphics library to draw multiple objects using this data with a single drawing call.

One of the solutions to this problem can be instancing, or instanced rendering. It is a way of executing the same drawing commands many times in a row, with each producing a slightly different result. This can be a very performant method of rendering a large amount of geometry with very few draw calls.

Another solution is to perform what is called frustum culling or clipping. The view frustum is the region of space in the modeled world that may appear on the screen, it is what the camera can see.

Frustum culling or clipping is the process of removing objects that lie completely outside the viewing frustum from the rendering process. Rendering these objects would be a waste of time since they are not directly visible and by not rendering them we can lower the amount of calls to the GPU.

Both of these solutions are not exclusive to each other and it is generally encouraged to apply both of them.

There are also other types of visibility culling, such as hierarchical depth culling; and there is a software called Umbra which is used in many AAA games that is used for discarding non-visible geometry. These methods try to avoid sending geometry to draw if they are ultimately not going to be visible.

In 3D scenes a technique called depth prepass helps with this as well, first the depth of the scene is drawn without color, then the depth buffer is used to remove the non-visible geometry with the help of visibility culling algorithms, and finally the visible geometry is drawn.

2.2.3 Algorithmic Complexity

When dealing with lots of objects in a scene that interact with each other, there are normally problems with efficiency. These problems can arise in the physics system for example, where checking for collisions can result in operations with O(N^2) cost.

Space partitioning[(W22)] algorithms are one way of solving this problem, it consists of the process of dividing a space into two or more disjoint subsets. In other words, space partitioning divides a space into non-overlapping regions and any point in the space can lie in exactly one of the regions.

Depending on how the data is structured we can have different types of partitions:

- **Grid:** A grid is a data structure that has equal sized cells. They are the simplest type of space partitioning and are most often used to partition 2D spaces.
- **Quadtrees:** A quadtree is a tree data structure in which each internal node has exactly four children. They are most often used to partition 2D spaces by recursively subdividing it into four quadrants or regions.
- **KD Trees:** A k-d tree (short for k-dimensional tree) is a space partitioning data structure for organizing points in a k-dimensional space. They are a special case of binary space partitioning trees.
- **BSP Trees:** Binary space partitioning (BSP) is a method for recursively subdividing a space into two convex sets by using hyperplanes as partitions. It was developed in the context of 3D computer graphics in 1969, and its structure is useful in rendering and performing geometrical operations among others.

3. Project Management

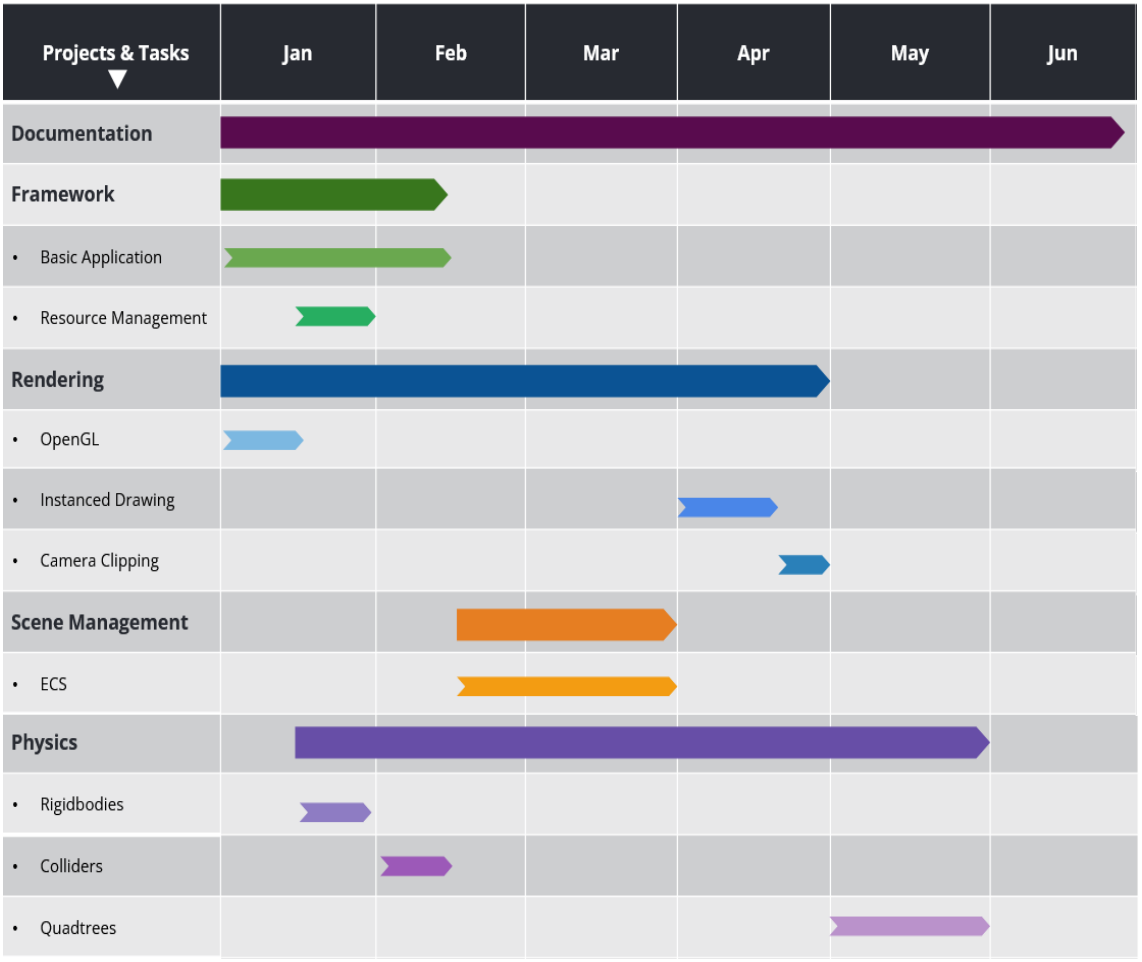
This section describes all the necessary aspects of the management of the project that help to plan and measure the progress of the tasks needed for the completion of the project.

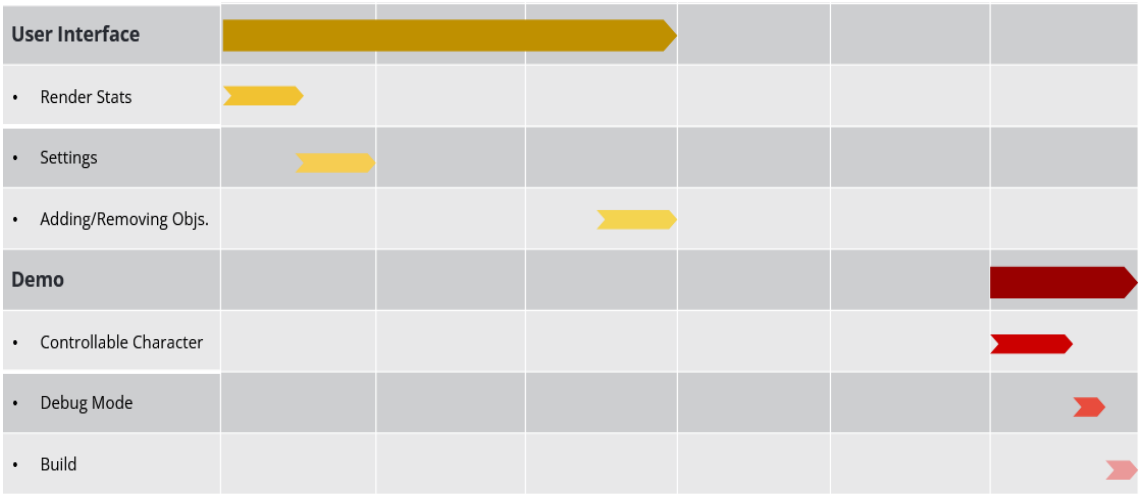
3.1 Procedure and Tools

3.1.1 Gantt

To organize and distribute all the work, a Gantt diagram is used. A Gantt diagram shows the assigned time for every task. This allows the measurement of the progress of the project and helps to compare it to the original plan and adapt or change objectives in consequence.

This diagram is represented in the following table.





T 3.1 - Gantt Table

3.1.2 Version Control

Since the beginning of the development, a repository was created in Github, each main step of the optimization has its own branch to keep code separated and make it easier to switch between versions.

Each branch is created at the start of the optimization's implementation and when it has been completed, a new branch will be created containing the new version.

3.2. SWOT Analysis

The SWOT analysis of the project can be broken down into:

Strengths

- Free open-source. Everyone can build on top of it.
- Step by step documentation of the project.
- Modular. Code is separated and can be easily updated or removed.

Weaknesses

- Solo developer. Only one person is working on the project.
- No visibility. There isn't a platform to share and gain visibility.

Opportunities

- There aren't many practical examples with documentation of the process.
- It is a useful base for creating tools.

Threats

- There are already complete engines widely used and with a lot of tools.
- Very small market sector. Not so many developers create their own technology because the vast majority tend to use a commercial engine.

3.3 Risks and Contingency Plans

This section analyzes all the risks taken into account in this project, and its possible solutions.

3.3.1 Lack of time

The main risk of the project is the lack of time, documenting all the process and performing tests each step adds up a lot of time to the development of the application.

By planning ahead and leaving some time at the end of each part to test and polish we can better organize the time needed to complete each task and minimize the risk of running out of time.

3.3.2 Lack of knowledge

The second risk is the lack of knowledge in some areas of the project that can result in taking extra time to learn about the concept and possible wrong implementations or bugs.

To avoid this problem, information was gathered about all the topics that were going to be implemented without previous experience. Also, the potential time deviation of the tasks related to these topics were taken into consideration.

3.3.3 Risk of the Tasks

The following table shows all of the tasks and the potential time deviation.

Task	Estimated Time (weeks)	Potential Deviation	Planned Time (weeks)
Framework	**6**	**Low**	**6**
Basic Application	5	Low	5
Resource Management	1	Low	1
Rendering	**6**	**Medium**	**7**
OpenGL	2	Low	2
Batch Rendering	3	Medium	4
Camera Clipping	1	Low	1
Scene Management	**6**	**High**	**9**
ECS	6	High	9
Physics	**5**	**Medium**	**6**
Rigidbodies	1	Low	1
Colliders	2	Low	2
Space Partitioning	2	Medium	3
User Interface	**6**	**Low**	**6**
Render Stats	1	Low	1
Settings	2	Low	2
Adding/Removing Objs	2	Low	2
Demo	**4**	**Low**	**4**
Debug Mode	1	Low	1
Build	1	Low	1

T 3.3 - Deviation of tasks

Some of the tasks have a higher potential deviation than others, these tasks are the following.

Batching - Medium
Batching is an important optimization for the performance of rendering lots of objects. The reason it is labeled as medium risk is because I have never implemented it before so it is a potential risk.

Entity Component System - High
The ECS is the part where the data-oriented design optimization will take part. As this includes a big change in how the code is structured and also implements an important optimization of the project, it is labeled as high risk.

Space Partitioning - Medium
A Fixed Resolution Grid is the space partitioning technique that is going to be implemented. It is the main optimization for the physics operations in the engine and is labeled as medium risk because it is an important part of the project and its implementation could potentially be a risk if something were to go wrong.

3.4 Costs Analysis

The following table shows all of the costs planned for the development of the project. For the cost estimates, we are going to assume a total number of 300 hours of work and that the cost per hour is 10€.

Type	Subject	Price	Type	Amortization (years)	Total Price
Direct Costs					
Personal	Salary / Time	3.000,00 €	Total	-	3.000,00 €
Equipment	Computer	1.200,00 €	Amortization	5	180,00 €
	Mouse	30,00 €	Amortization	3	11,25 €
Consumables	Paper sheets	3,00 €	Unique	-	3,00 €
	Pencils	3,00 €	Unique	-	3,00 €
Software	Visual Studio	0,00 €	Monthly	-	0,00 €
	Github	0,00 €	Monthly	-	0,00 €
	HacknPlan	0,00 €	Monthly	-	0,00 €
Indirect Costs					
Maintenance	Electricity	20,00 €	Monthly	-	120,00 €
	Water	12,00 €	Monthly	-	72,00 €
	Food	40,00 €	Monthly	-	240,00 €
Total:					**3.629,25 €**

Project Duration	6 Months

T 3.4 - Costs Analysis

As we can see, the main cost is of human resources, in software development projects the necessary material is minimal and the software is generally accessible free of charge for students.

Indirect costs are calculated approximately.

3.5 Changes in Planification

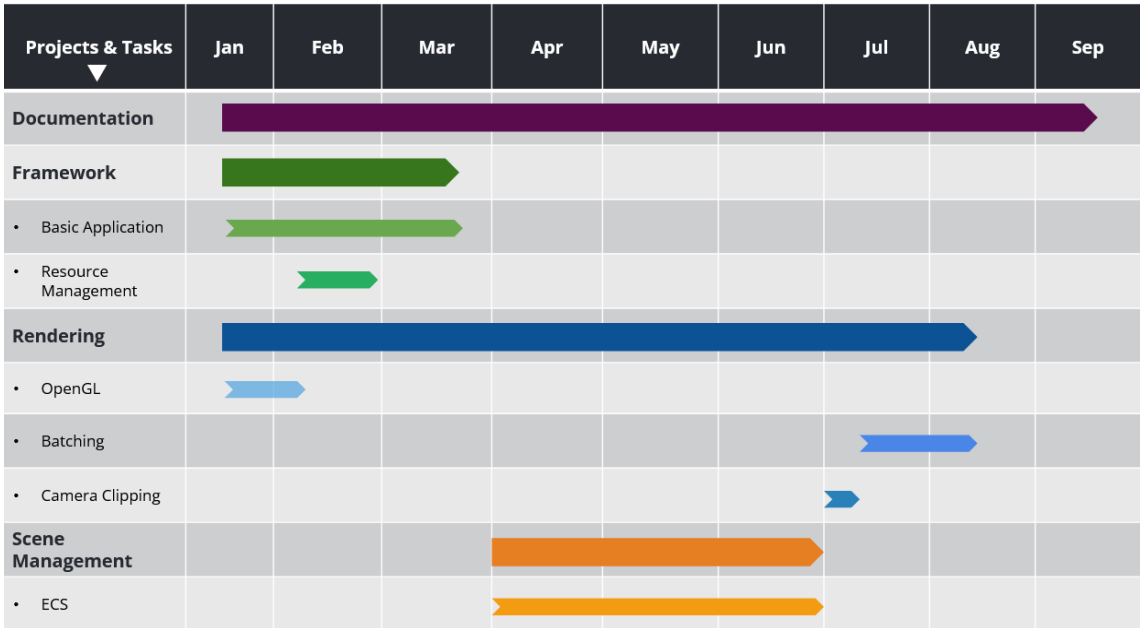
In this section we will talk about the changes on the initial plan for the development of the project and how I have adapted to solve them.

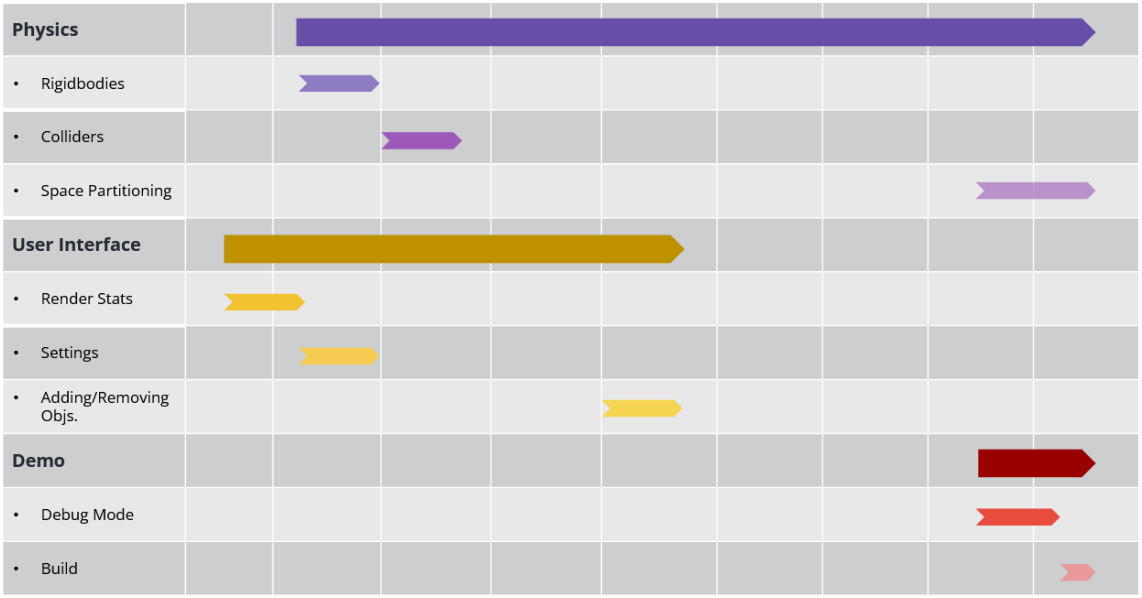
The main reason for the delay of the finish date was lack of time due to not correctly predicting the time it would take certain tasks. This misprediction was mostly because of unexpected problems and bugs that appeared and were very time-consuming.

I also failed to predict the time needed for testing and profiling and it contributed to this lack of time too.

The way I adapted to these problems was to extend the deadline and cut the playable demo. Instead of a playable demo there will be a basic application where you can modify some world parameters and add and remove asteroids.

The following Gantt diagram reflects the adapted times for each task.





T 3.5.1 - Adapted Gantt Diagram

As the Gantt diagram shows, the project was delayed 2,5 months until the middle of September. This extension in time also implies additional costs to the development of the project, we can see the new costs analysis table below.

Type	Subject	Price	Type	Amortization (years)	Total Price
Direct Costs					
Personal	Salary / Time	4.500,00 €	Total	-	4.500,00 €
Equipment	Computer	1.200,00 €	Amortization	5	180,00 €
	Mouse	30,00 €	Amortization	3	11,25 €
Consumables	Paper sheets	3,00 €	Unique	-	3,00 €
	Pencils	3,00 €	Unique	-	3,00 €
Software	Visual Studio	0,00 €	Monthly	-	0,00 €
	Github	0,00 €	Monthly	-	0,00 €
	HacknPlan	0,00 €	Monthly	-	0,00 €
Indirect Costs					
Maintenance	Electricity	20,00 €	Monthly	-	180,00 €
	Water	12,00 €	Monthly	-	108,00 €
	Food	40,00 €	Monthly	-	360,00 €
Total:					**5.345,25 €**

Project Duration	9 Months

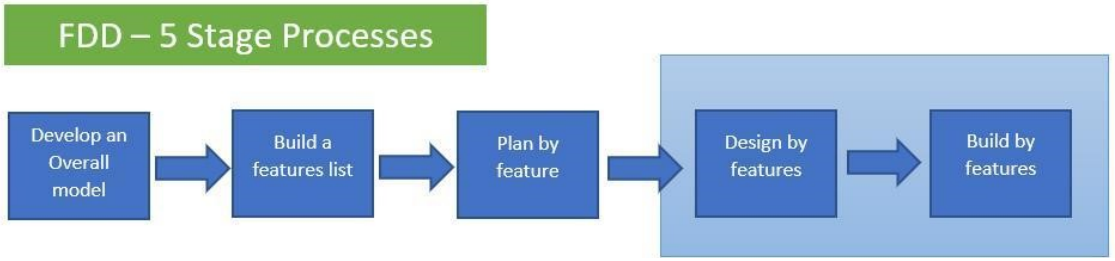
T 3.5.2 - Updated Costs

4. Methodology

For this project, an Agile methodology called Feature-Driven Development (FDD) was used to track the progress of the project and its different parts.

4.1 Feature-Driven Development[W7]

As we can see in the following figure, Feature-Driven Development defines five steps in two phases: planning and construction.



F 4.1 - Feature-Driven Development diagram

4.1.1 Planning Phase

The planning phase is all about getting an accurate understanding of content and context of the project.

- Develop an Overall Model refers to clarifying what the final objective is. What things need to be done and how they can be approached.
- Build a Features List is to lay out all the features the final result will have. These features should be purposes or smaller goals, rather than tasks.
- Plan by Feature refers to organizing all the features you need to implement in a prioritized list, taking into account dependencies between features and importance of each feature in the overall project.

4.1.2 Construction Phase

In the design and build phase of FDD, you work through the feature list on a feature-by-feature basis so that when all the features are implemented, the result would be a finished product. Each feature is developed in two steps.

- Design by Feature is the step where the feature is designed and every task needed for the implementation is layed out.
- Build by Feature is the step where the actual implementation occurs. After the feature is implemented and tested, if the result is positive, the construction phase begins for the next feature.

4.1.3 Feature-Driven Development in this Project

The way this methodology was applied to the project was to divide it into four versions, with defined objectives and features to implement.

The planning phase consisted in exactly this, listing all the necessary features for each version and sorting them by priority.

The construction phase consisted in implementing all the features needed for each version. After implementing them, tests were performed to ensure a positive result and that every version was better than the previous one, which meant that the construction phase for the next version could begin.

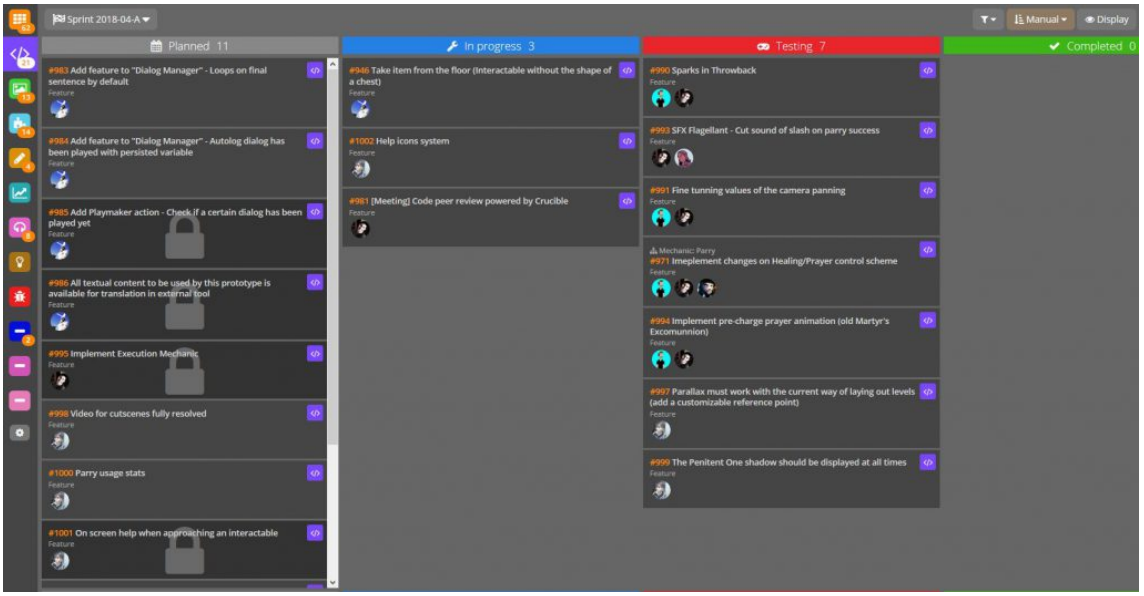
This process was repeated until the completion of every version.

4.2 Tracking Tools

To track the development of the project, a tool named HacknPlan was used along the Gantt diagram shown previously.

HacknPlan[(W8)] allows tracking tasks and their completion in a useful way to help with the development of each feature implemented.

The following figure shows an example of the interface of HacknPlan.



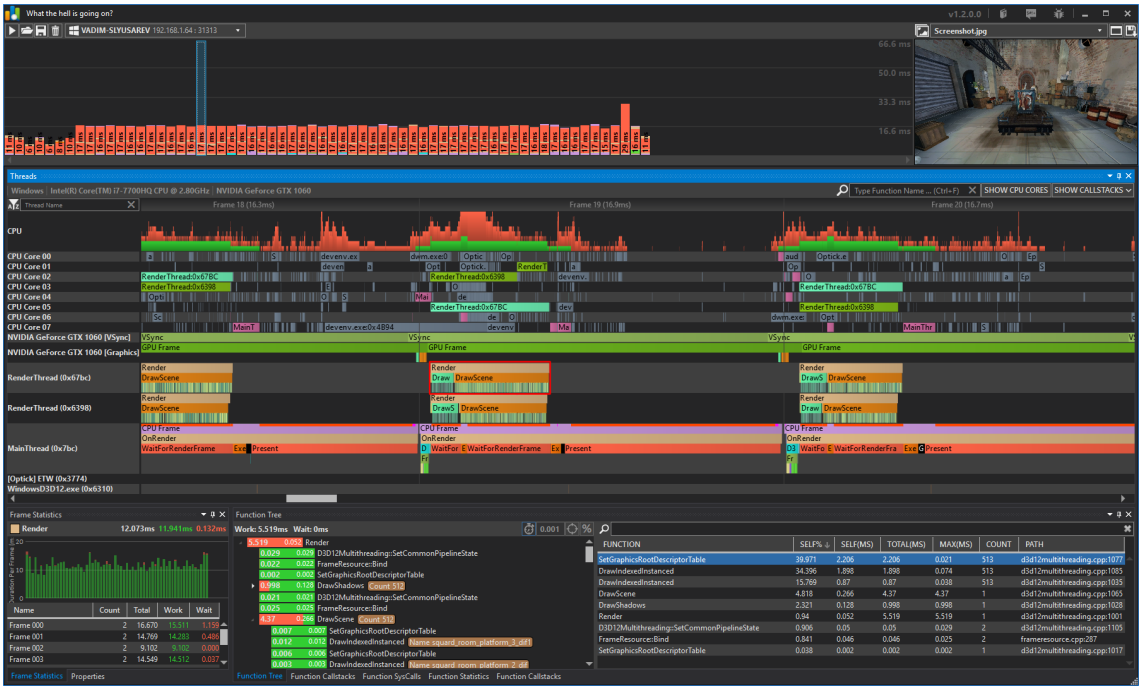
F 4.2 - HacknPlan interface

4.3 Validation Tools

In order to validate the completion of the tasks, when a new implementation is finished, a series of tests are done to ensure correct performance.

These tests consist of profiling each iteration focusing on specific parts of the application where the optimizations take place and comparing them with previous iterations.

In order to do the profiling of the project, a tool named Optick was used. In the following figure we can see a screenshot of the application.



F 4.3 - Optick interface

Regular tests are also done during the development of the project to check for bugs or small errors.

After a new iteration has passed these tests, a meeting with the director is arranged to show the advancements and, if it is validated by the director, start working on the next iteration.

5. Development

In this section we will explain the basic application and analyze each version that is implemented during the development of the project.

The project is divided into four different versions where a specific optimization or group of optimizations is implemented, each version is tested against the previous one to ensure the correct improvement of the project and the completion of the objectives set at the start of the development.

5.1 Environment

All the code is written in C++ with VisualStudio compiler which allows for easy and fast debugging.

The operative system chosen is Windows and the main external libraries are the following:

- **OpenGL**[W10]**:** stands for Open Graphics Library. It is a cross-language, cross-platform application programming interface (API) for rendering 2D and 3D vector graphics.
  It was released on June 30, 1992 by Silicon Graphics, Inc. and, since 2006, has been managed by Khronos Group.

- **SDL**[W11]**:** stands for Simple DirectMedia Layer. It is a cross-platform software development library designed to provide a hardware abstraction layer for computer multimedia hardware components. SDL manages video, audio, input devices, CD-ROM, threads, shared object loading, networking and timers. For this project we are only using it for the input manager and to create the window.

- **GLEW**[W12]**:** stands for OpenGL Extension Wrangler Library. It is a cross-platform open-source C/C++ extension loading library. It provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform.

- **Dear ImGui**[W13]**:** is a bloat-free graphical user interface library for C++. It is fast, portable, renderer agnostic and self-contained (no external dependencies). It was used for creating the UI panel.

- **PCG Random**[W14]**:** PCG is a family of simple fast space-efficient statistically good algorithms for random number generation.
- **GLM**[W15]**:** stands for OpenGL Mathematics. It is a header-only C++ mathematics library for graphics software based on the OpenGL Shading Language (GLSL) specifications. It provides classes and functions designed and implemented with the same naming conventions and functionalities as GLSL.
- **stb_image**[W16]**:**is a group of single-file public domain (or MIT licensed) libraries for C/C++. This project uses the stb_image library for loading and drawing images.
- **Optick**[W9]**:** is a super-lightweight C++ profiler for Games. It provides access for all the necessary tools required for efficient performance analysis and optimization: instrumentation, switch-contexts, sampling, GPU counters. It was used for profiling each version of the project.

5.2 The Application

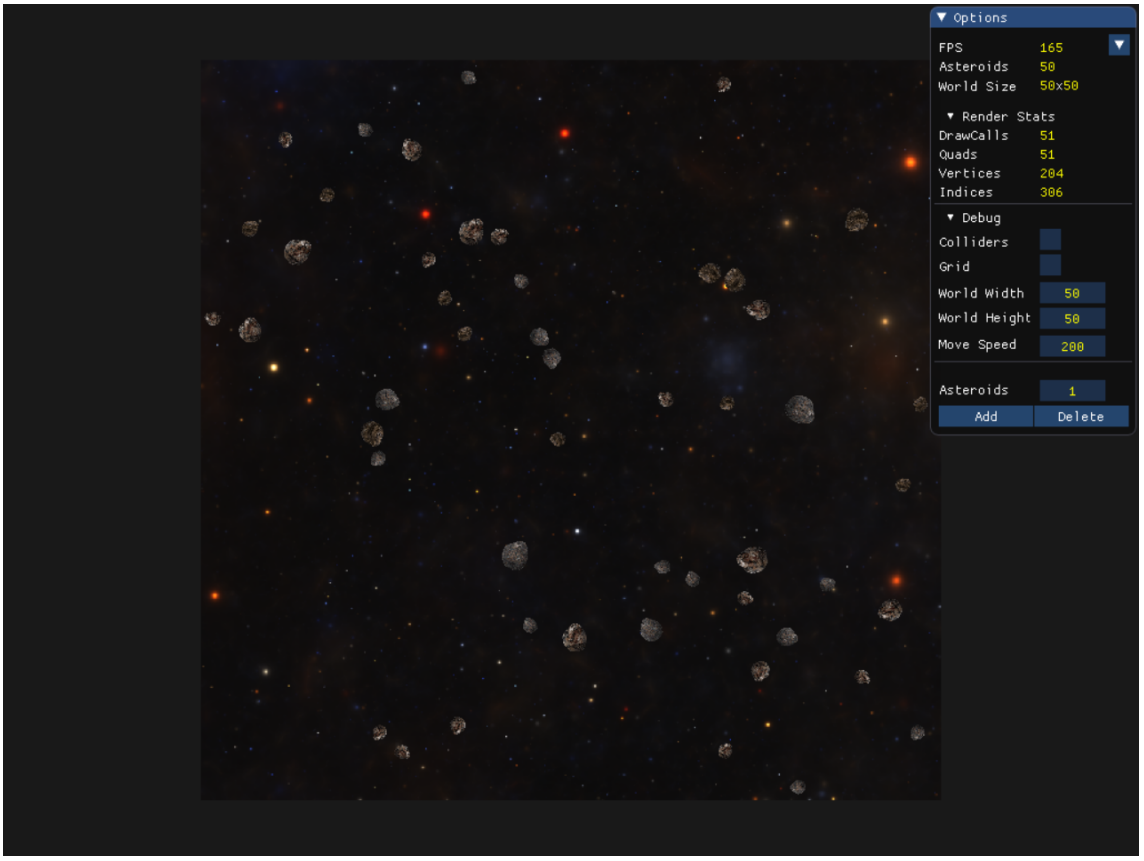
Every version is basically the same application but with optimizations built on top of the previous one.

The application itself is a minimal 2D engine that allows the user to add and remove asteroids from the scene. These asteroids collide and bounce off each other and the edges of the background.

The camera can be zoomed in and out with the mouse wheel and, if the map is not shown completely, the camera can be moved with the WASD keys.

There is a UI panel that shows some stats, such as the framerate or the number of draw calls. It also has sliders to tweak some aspects of the scene like the background size or the camera speed. There are also two checkboxes to show a grid, and the debug view of the colliders.

The following figure shows a screenshot of the application where we can see the background, the asteroids and the UI panel.



F 5.2 - Application screenshot

5.3 Version 1 - Object Oriented

This is the first version of the project, it is the application coded with an object oriented approach. This version serves as the foundation and the optimizations of every version are applied gradually on top of this initial version.

5.3.1 Object Oriented Programming[W17]

Object Oriented Programming (OOP) is a programming paradigm based on the concept of “objects”, which can contain data and code: data in the form of fields (often known as “attributes” or “properties”), and code, in the form of procedures (often known as “methods”).

OOP is structured in the following way:

- **Classes:** user-defined data types that act as the blueprint for individual objects, attributes and methods.
- **Objects:** instances of a class located in the main memory and created with specifically defined data. Objects can correspond to real-world objects or an abstract entity.
- **Methods:** functions that are defined inside a class that describe the behaviors of an object. Each method contained in class definitions starts with a reference to an instance object. Additionally, the subroutines contained in an object are called instance methods. Programmers use methods for reusability or keeping functionality encapsulated inside one object at a time.
- **Attributes:** are defined in the class template and represent the state of an object. Objects will have data stored in their attributes.

The main principles of OOP are:

- **Encapsulation:** all the important information is contained inside an object and only select information is exposed. The implementation and state of each object are privately held inside a defined class. Other objects do not have access to this class or the authority to make changes. They are only able to call a list of public functions or

methods. This characteristic of data hiding provides greater program security and avoids unintended data corruption.

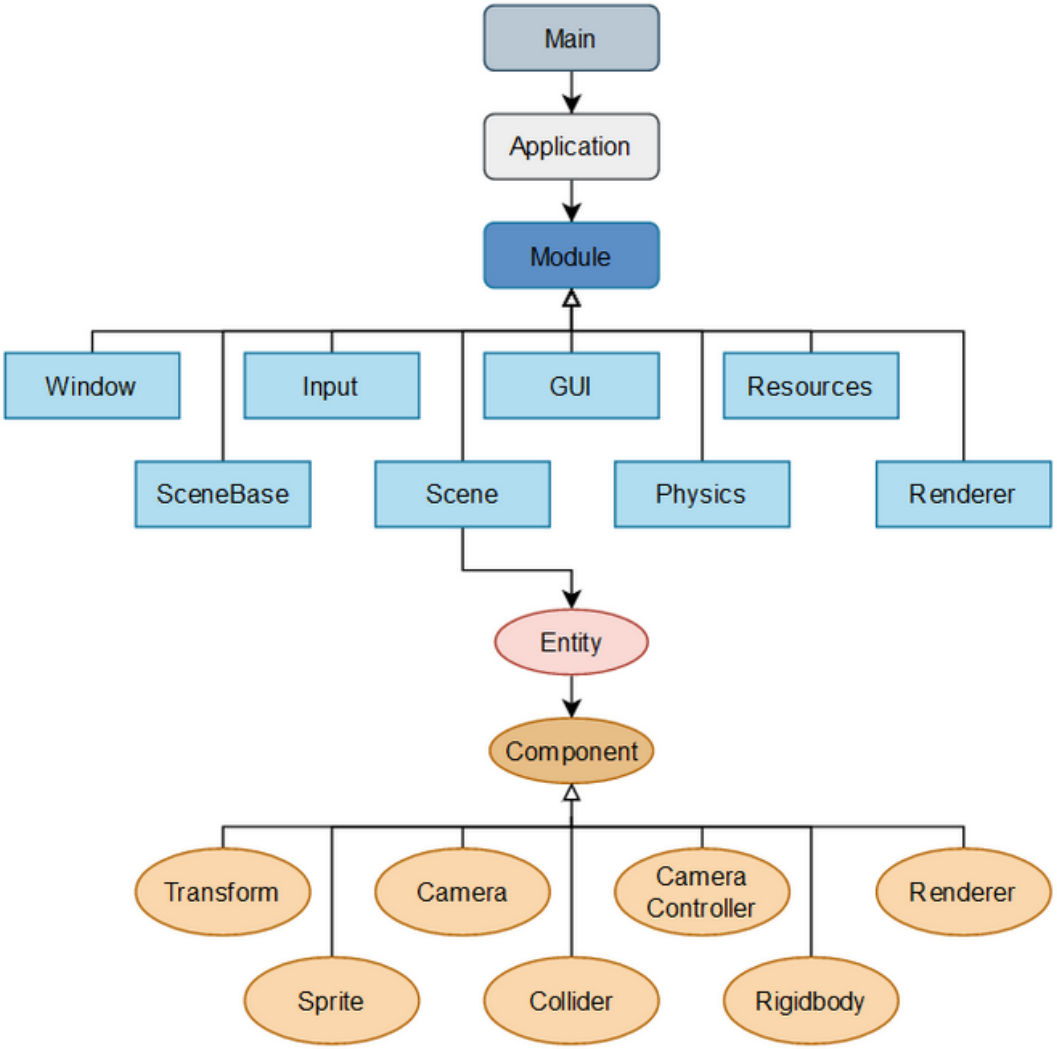
- **Abstraction:** objects only reveal internal mechanisms that are relevant for the use of other objects, hiding any unnecessary implementation code. The derived class can have its functionality extended. This concept can help developers to more easily make additional changes or additions over time.

- **Inheritance:** classes can reuse code from other classes. Relationships and subclasses between objects can be assigned, enabling developers to reuse common logic while still maintaining a unique hierarchy. This property of OOP forces a more thorough data analysis, reduces development time and ensures a higher level of accuracy.

- **Polymorphism:** objects are designed to share behaviors and they can take on more than one form. The program will determine which meaning or usage is necessary for each execution of that object from a parent class, reducing the need to duplicate code. A child class is then created, which extends the functionality of the parent class. Polymorphism allows different types of objects to pass through the same interface.

5.3.2 Application Structure

The way in which the base engine is structured is shown in the following diagram.



F 5.3.2 - Application Structure in OOP

Main is the main file that contains the main loop, and Application is the base class that contains all the modules and performs operations regarding framerate and modules' management.

We will now explain each one of the parts of this diagram.

5.3.3 Modules

A module is a part of the code that is in charge of a specific aspect of the application. They contain all the necessary functions and data to manage the operations and data they need for their task.

The following figure shows the base Module class.

```cpp
#define MODULE_NAME_LENGTH 25

class Module
{
public:
    Module(const char* name, bool start_enabled = true) : enabled(start_enabled) { strcpy_s(this->name, MODULE_NAME_LENGTH, name); }
    virtual ~Module() {}

    virtual bool Init() { return true; }
    virtual bool Start() { return true; }
    virtual bool PreUpdate(float dt) { return true; }
    virtual bool Update(float dt) { return true; }
    virtual bool PostUpdate(float dt) { return true; }
    virtual bool CleanUp() { return true; }

    const char* GetName() const { return name; }
    bool IsEnabled() const { return enabled; }

    void SetEnabled(bool active)
    {
        if (enabled != active)
        {
            enabled = active;
            if (active == true)
                Start();
            else
                CleanUp();
        }
    }

private:
    bool enabled;
    char name[MODULE_NAME_LENGTH];
};
```

F 5.3.3 - Module class in OOP

The list of modules in the project are:

- **ModuleInput:** manages all input related operations.
- **ModuleWindow:** manages the window.
- **ModuleRenderer:** contains all rendering related code.
- **ModuleGUI:** is in charge of drawing and managing the UI.
- **ModuleScene:** manages the scene of the application, it contains and manages all the objects inside the scene.
- **ModuleSceneBase:** is in charge of the camera and the debug options.
- **ModulePhysics:** contains all the operations related to physics simulation.
- **ModuleResources:** manages the resources and the operations regarding loading, unloading and storing them. There are only two types of resources in the application: textures and shaders.

5.3.4 Objects

The class Entity is the base class for every object, it contains a list of components and has the operations regarding components' creation and deletion.

The following figure shows the Entity header file.

```
class Entity
{
public:
    Entity() {};
    virtual ~Entity();

    UID GetUID() { return uid; }
    const std::vector<Component*>& GetComponents() const { return components; }

    Component* AddComponent(Component::Type component);
    void DeleteComponent(Component::Type component);
    Component* GetComponent(Component::Type component) const;

    void Draw();

private:
    std::vector<Component*> components;
    UID uid = 0;
};
```

F 5.3.4 - Entity class in OOP

5.3.5 Components

Components are the building blocks that conform the entities, they contain the data and operations to perform the task needed.

An entity or object can have different combinations of components depending on what they are going to be used for.

The following figure shows the base class inherited by every component.

```
class Component
{
public:
    enum class Type
    {
        TRANSFORM,
        CAMERA,
        CAMERA_CONTROLLER,
        RENDERER,
        SPRITE,
        COLLIDER,
        RIGIDBODY,
        UNKNOWN
    };

public:
    Component(Component::Type type, Entity* entity) : type(type), entity(entity) {}
    virtual ~Component() {};

    Component::Type GetType() const { return type; }
    Entity* GetEntity() const { return entity; }

    virtual void OnUpdate(float dt) {};

private:
    mutable Component::Type type;
    Entity* entity = nullptr;
};
```

F 5.3.5 - Component class in OOP

Depending on the type we can have the following components:

- **Transform:** manages the position, rotation, and scale of the object.
- **Camera:** manages the position and rotation of the camera object and also the view, projection and viewProjection matrices.
- **Camera Controller:** manages the movement of the camera, the zoom and the speed in which it moves.
- **Renderer:** draws the entity.
- **Sprite:** holds the texture to be drawn and the shader to be used.
- **Collider:** can be a Circle Collider or a Rectangle Collider, the borders of the scene are rectangle colliders while the asteroids have circle colliders. It manages the position and size of the object's collider.

- **RigidBody:** manages the velocity, mass and rotational speed of the object. Along with the collider component they are used by the Physics Module to perform physics operations.

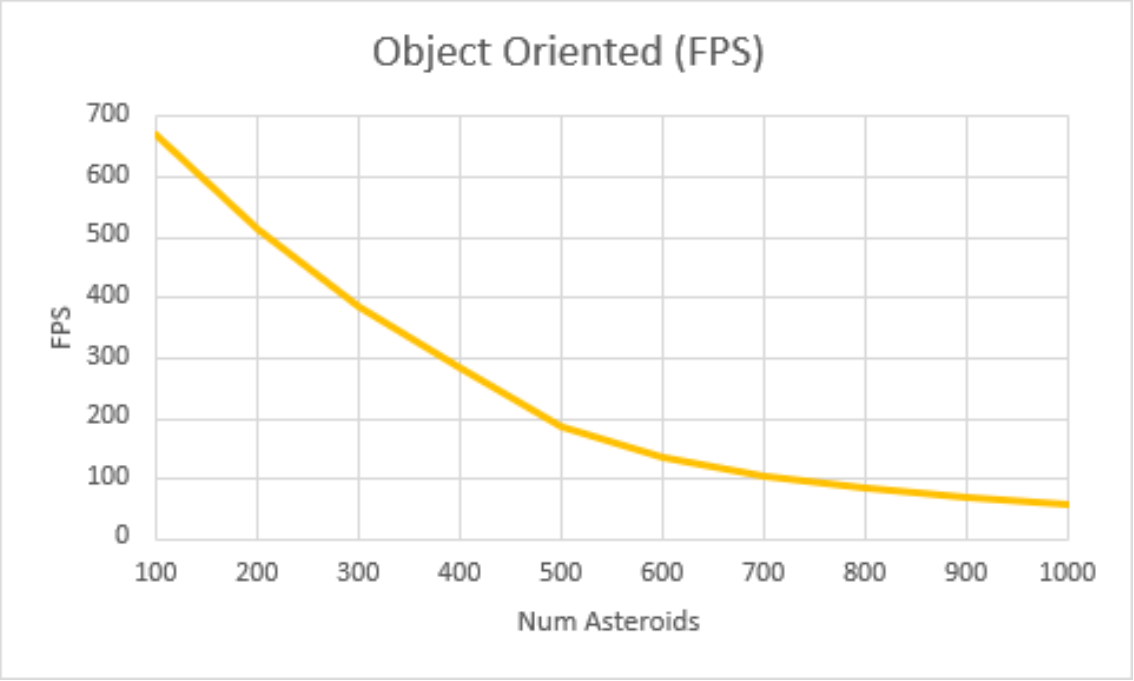
5.3.4 Profiling

After finishing this version, profiling was done with the help of Optick. The key aspects that we are focusing on every version in this version are:

- Frame Rate (FPS)
- Main Update function
- Draw function
- Physics calculations

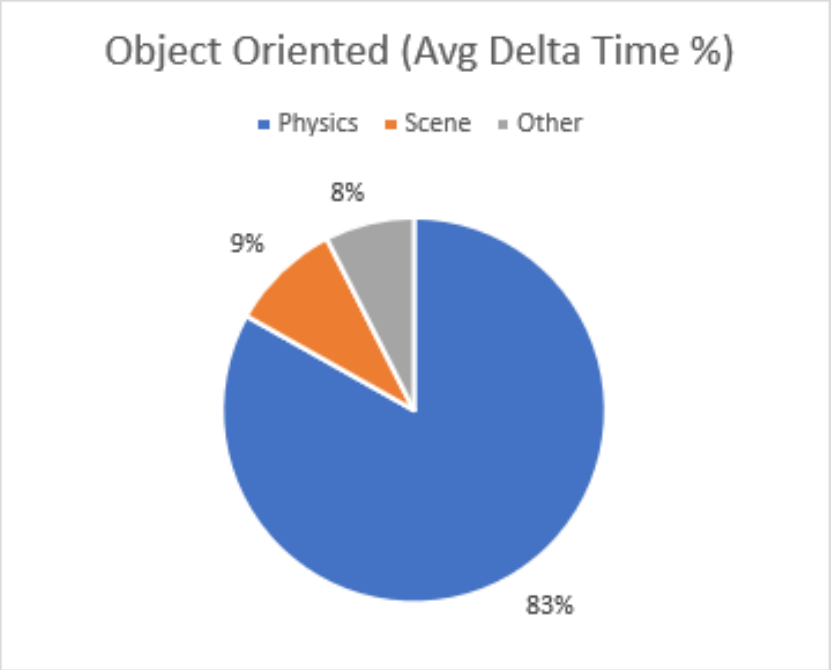
In the next versions we will use the following graphs to make comparisons and check if the optimizations implemented translate to a better performance.

The following graph shows how the frames-per-second (FPS) vary when adding more and more entities. The limit is set to 1000 asteroids because that is when the FPS become lower than 60 which is the minimum frame rate we want to maintain.



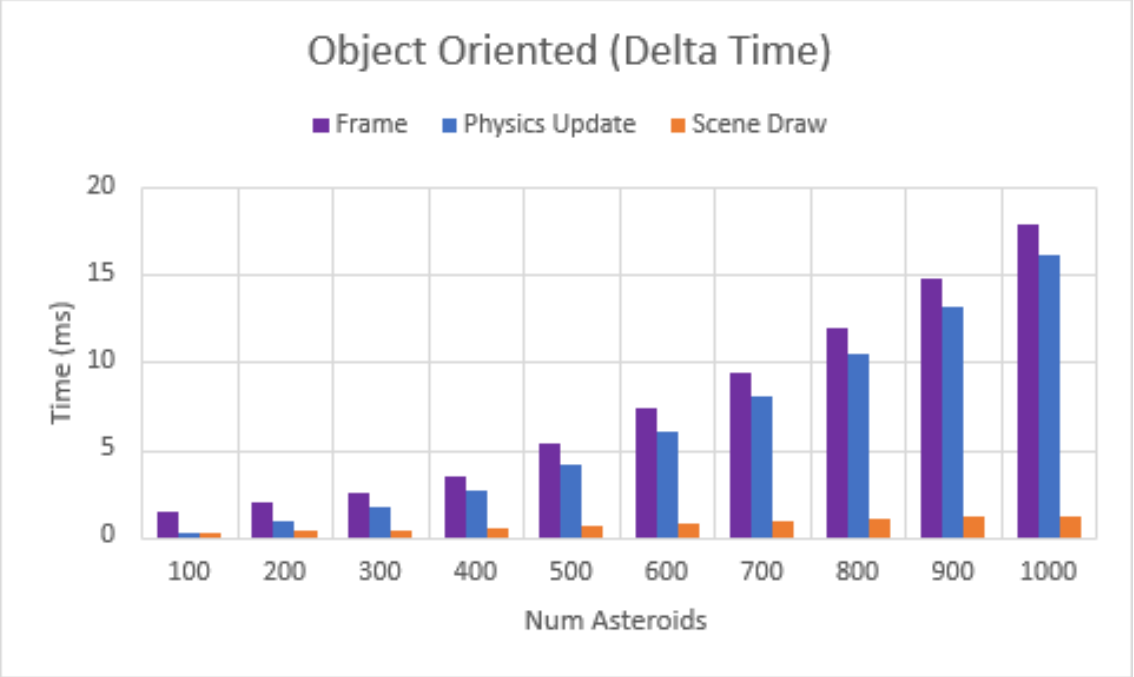
F 5.3.4.1 - Object Oriented (FPS) graph

This graph shows the overall percentages of the time it takes to process a frame. We can see that the physics calculations are clearly what takes the most time.



F 5.3.4.2 - Object Oriented (Avg. Delta Time %) graph

The next graph shows the milliseconds it takes to compute each part, and how it grows the more asteroids we have.



F 5.3.4.3 - Object Oriented (Delta Time) graph

5.4 Version 2 - Data Oriented

In this section we will analyze the development of the second version of the project.

5.4.1 Data Oriented Design

Data-Oriented Design shifts the perspective of programming from objects to the data itself: The type of the data, how it is laid out in memory, and how it will be read and processed.[W18]

If we look at a program from the data point of view, the ideal data generally is in a format that uses the least amount of transformations. Very often this ideal data layout will be large blocks of contiguous, homogeneous data that can be processed sequentially.

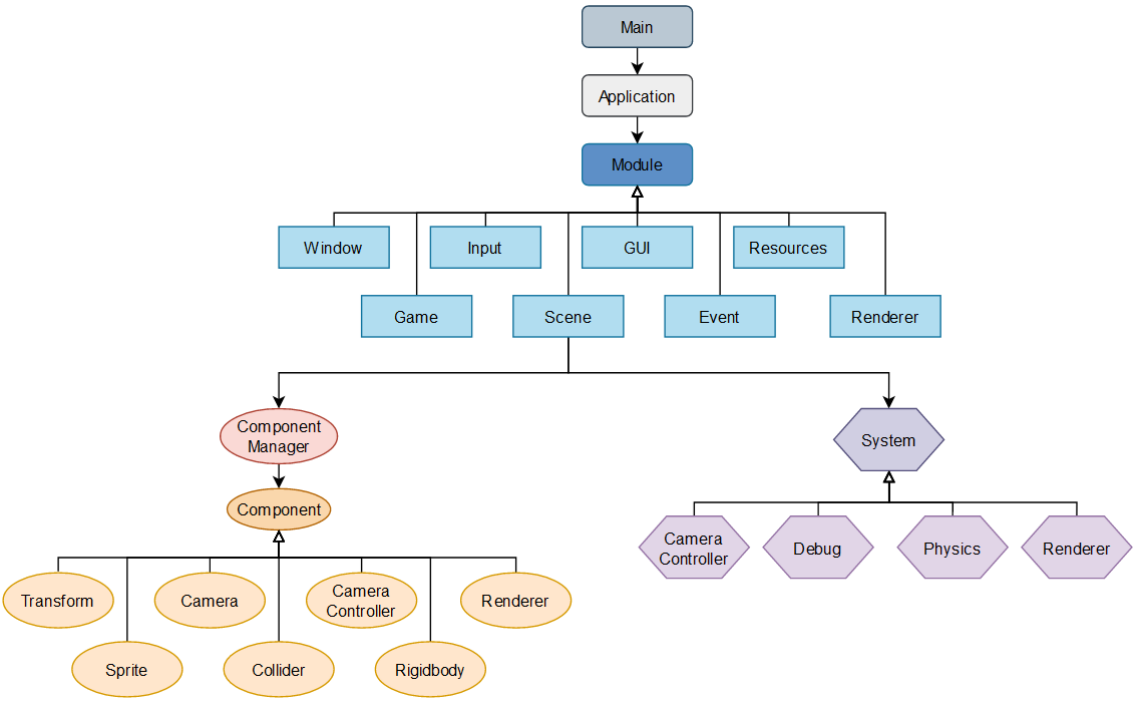
When we think about objects, we immediately think of some form of trees and our data is naturally arranged that way. As a result, when we perform an operation on an object, it will usually result in that object in turn accessing other objects further down in the tree. Iterating over a set of objects performing the same operation generates cascading, totally different operations at each object.

To achieve the best possible data layout, it's helpful to break down each object into the different components, and group components of the same type together in memory, regardless of what object they came from. This organization results in large blocks of homogeneous data, which allow us to process the data sequentially.[W19]

5.4.2 Application Structure

This version included some major changes to the core structure of the application. It was a complete rework of how the entities' data is stored, accessed and modified, as well as the code regarding components.

The structure of this version is shown in the following diagram.



F 5.4.2 - Application structure in DOD

We can see that there are two new modules: Game and Event. ModuleGame manages all the entities that are created in the scene and ModuleEvent manages the events sent internally in the engine.

The following sections explain the internal functioning of this version.

5.4.3 Entity Component System

In this version entities are just an ID number and a bitmask to find out which components an entity has. This can be seen in the next figure.

```
// --- Entities ---
std::array<ComponentMask, MAX_ENTITIES> component_masks;

std::queue<EntityIdx> available_indexes;
uint32_t count_entities = 0;
```

F 5.4.3.1 - Entities data in DOD

In order to have unique IDs, when an entity is created the ID is retrieved from a list of numbers and when it is destroyed its ID is returned to the list to be used by another future entity.

Components become just containers of data, as we can see in the figure below.

```
struct C_RigidBody
{
    glm::vec2 velocity = glm::vec2(0.0f);
    float mass = 0.0f;
    float rotation_speed = 0.0f;
};
```

F 5.4.3.2 - Component data in DOD

Components are stored in a big array for each type. To keep track of the data a sparse-set[W20] is used, it consists of two arrays: dense and sparse.

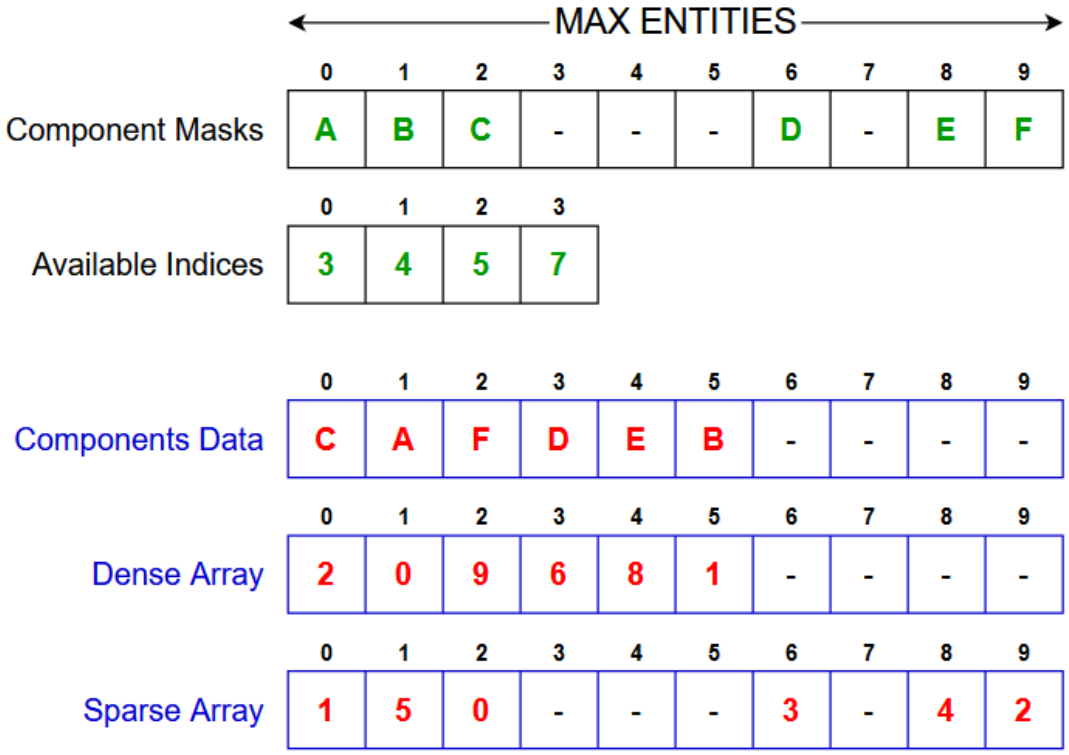
```
std::array<Component, MAX_ENTITIES> components;

std::array<uint32_t, MAX_ENTITIES> dense;
std::array<uint32_t, MAX_ENTITIES> sparse;
uint n = 0;
```

F 5.4.3.3 - Component Arrays data in DOD

The dense array is aligned with the components' array and the sparse array is aligned with the entities ID array (available_indexes).

'n' is just a counter of the number of components inside the array.

The following figure shows a diagram of how the data is stored and linked.



F 5.4.3.4 - Data Layout diagram

When creating a component, its data is stored in the last position of the array.

In the dense array we store an index to the sparse array, and in the sparse array we store the index to the dense array so that we can access each other from any of the two. After doing this we update the counter.

This is shown in the following figure.

```
void AddComponent(EntityIdx entity, Component& component)
{
    if (entity >= MAX_ENTITIES)
        return;

    components[n] = component;
    dense[n] = entity;
    sparse[entity] = n;
    ++n;
};
```

F 5.4.3.5 - AddComponent function in DOD

When removing an entity we first check if the entity has that component. Then we decrease the counter and overwrite the data of the component to be deleted with the data of the component in the back of the array.

The function to remove a component looks like this.

```
void RemoveComponent(EntityIdx entity)
{
    if (HasComponent(entity))
    {
        --n;
        Component data = components[n];
        uint32_t item = dense[n];
        uint32_t dense_index = sparse[entity];

        components[dense_index] = data;
        dense[dense_index] = item;
        sparse[item] = dense_index;
    }
}
```

F 5.4.3.6 - RemoveComponent function in DOD

Because components are just data, we need some way of transforming that data. Systems are the ones in charge of this.

Every system has a unique mask that tells which components an entity needs to have to be included in it, as well as an array to store those entities.

Each system has then an array for every type of component it works on to have the data stored locally.

The following figure shows the Renderer System that works on entities that have a transform, renderer and sprite components. If an entity has these three components but has more it will also be added to the system, but if it has just one or two of them it will not be added.

```
class S_Renderer : public System
{
public:
    S_Renderer();
    ~S_Renderer();

    void Update(float dt) override;

    void Render() override;

private:
    std::vector<C_Transform> transforms;
    std::vector<C_Renderer> renderers;
    std::vector<C_Sprite> sprites;
};
```

F 5.4.3.7 - System Renderer class in DOD

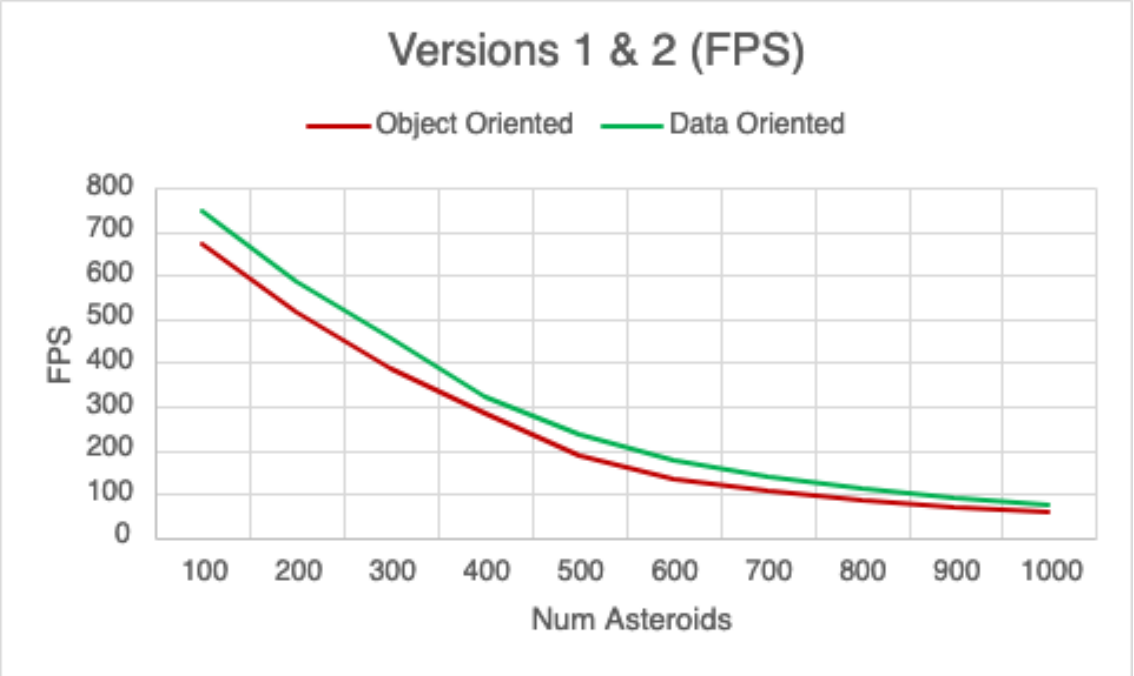
We have four systems inside the project:

- **Renderer:** is in charge of sending the necessary data to the ModuleRenderer which holds the necessary functions to draw every entity.
- **Debug:** is in charge of sending the data of the colliders of every entity to be drawn by the ModuleRenderer.
- **Camera:** manages the camera movement, zooming and resizing.
- **Physics:** manages all the collisions between entities and performs the necessary operations to simulate them.

5.4.4 Profiling

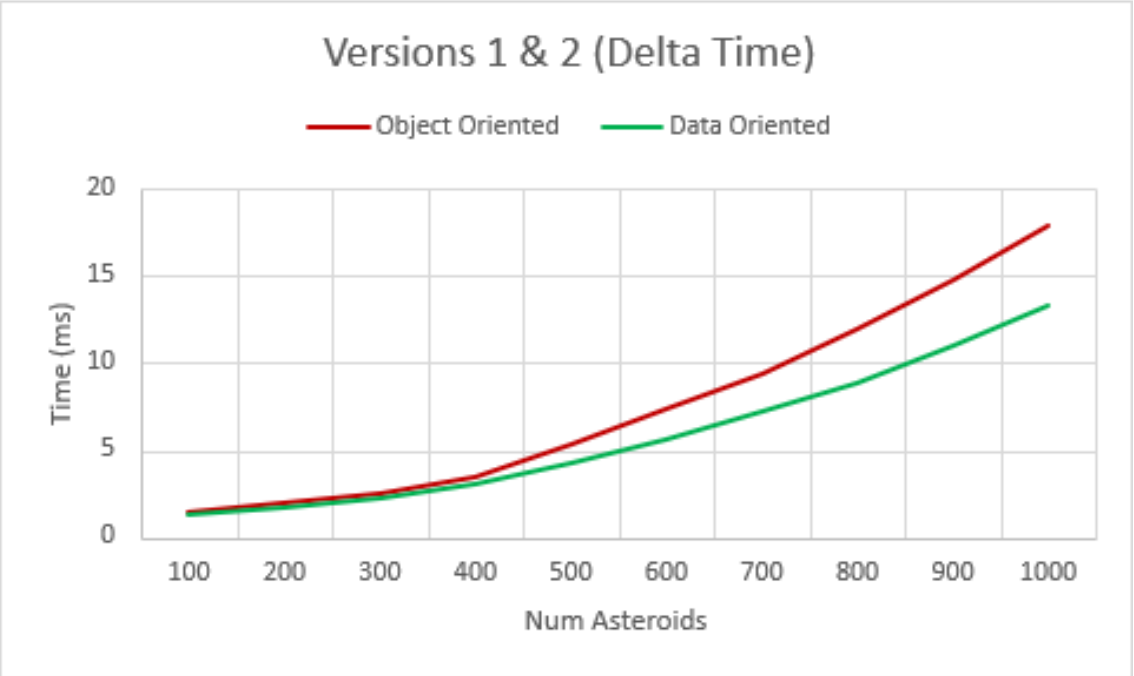
The tests performed after finishing this version followed the same methodology as the previous version.

The following graph shows the difference in performance between this version and the previous one.



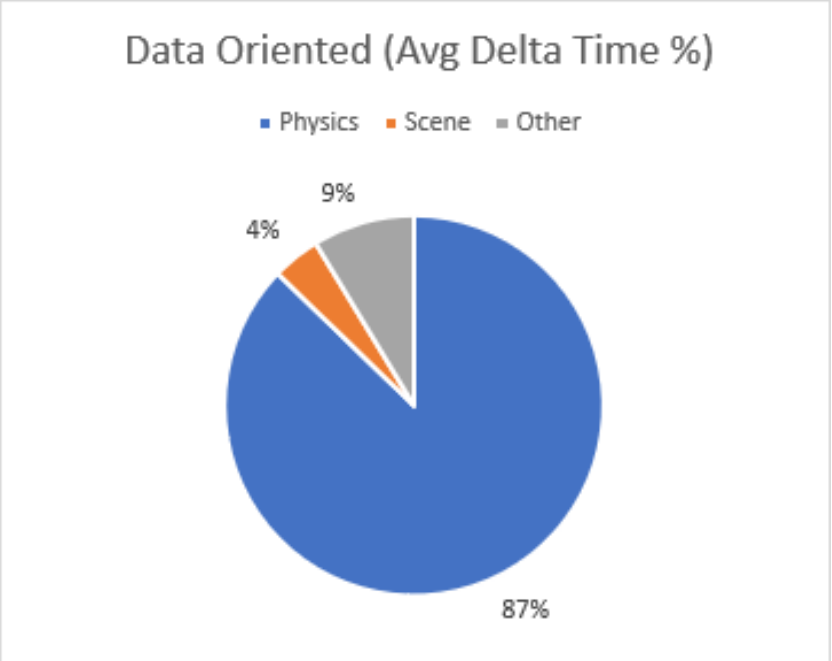
F 5.4.4.1 - Versions 1 & 2 (FPS) graph

The next graph shows the comparison between the delta time of both versions, we can see that as we have more asteroids, the object-oriented version takes more time for every frame than the data oriented-one. And, as we continue adding more asteroids, the difference between both of them increases.



F 5.4.4.2 - Versions 1 & 2 (Delta Time) graph

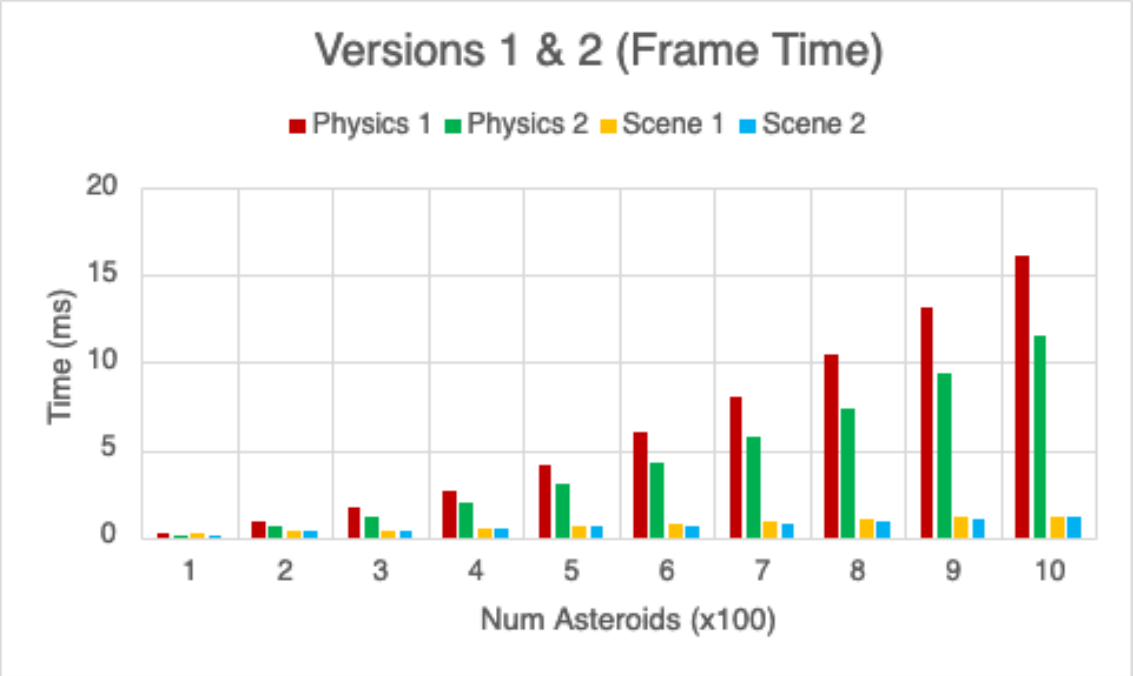
The graph below shows the overall percentages of the time it takes to process a frame. We can see the scene takes less time proportionally than in the previous version's 8% and that the physics calculations take up the extra percentage while the other calculations remain as 9%.



F 5.4.4.3 - Data Oriented (Avg Delta Time %) graph

The following graph shows the comparison between both versions and the time it takes each part of the frame.

We can see that the scene operations take more or less the same time, the new version being a little bit faster. The physics calculations are clearly more efficient in the new version.



F 5.4.4.4 - Versions 1 & 2 (Frame Time) graph

5.5 Version 3 - Rendering Optimizations

In this section we will focus on the third version of the project which consists of two rendering optimizations: frustum culling and batch rendering.

5.5.1 Frustum Culling

Frustum Culling or Camera Clipping is a rendering optimization with the primary focus of lowering the amount of data sent to the GPU.

It does this by checking if an object to be rendered is inside the camera boundaries, in other words if it will actually be seen. If the object is outside the camera, its data will not be drawn.

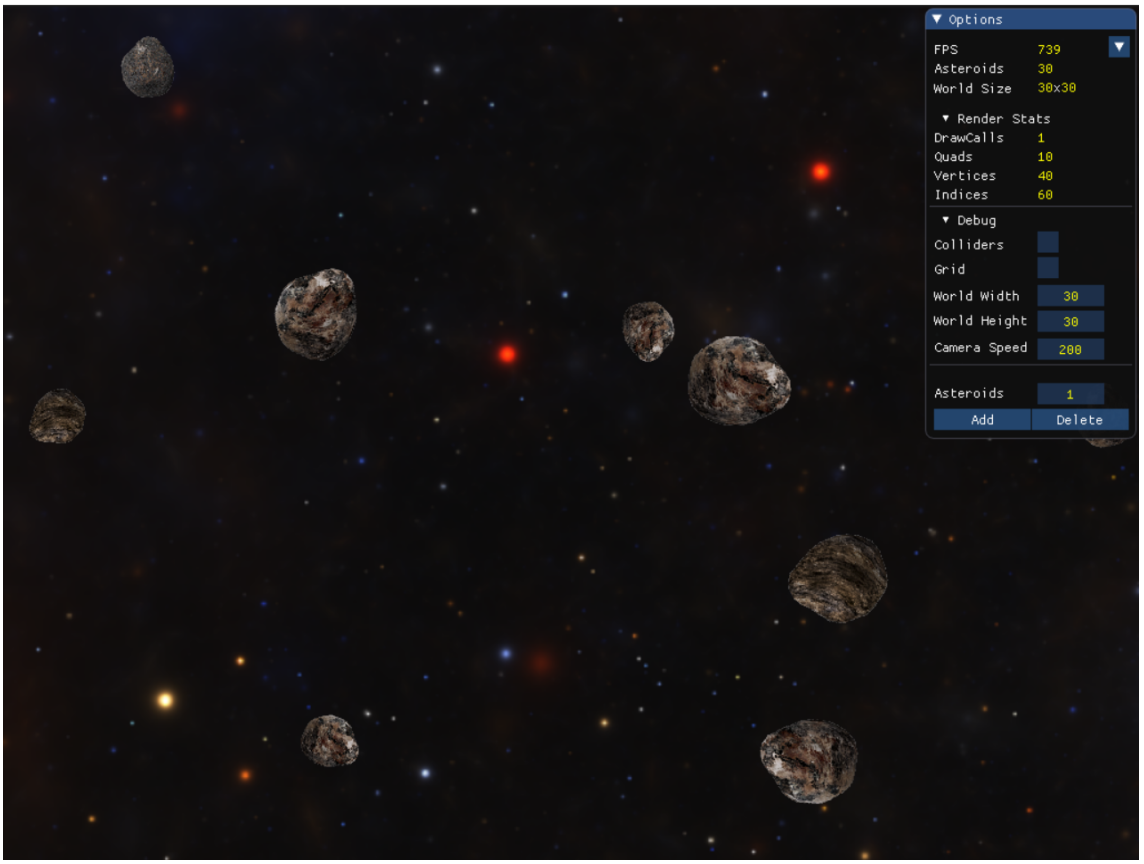
The following image shows how it is implemented in the project.

```
const bool ModuleGame::IsInsideCamera(C_Transform transform) const //--- For frustum culling
{
    float zoom = App->scene->GetComponent<C_CameraController>(main_camera).zoom;
    glm::vec2 camera_pos = App->scene->GetComponent<C_Transform>(main_camera).position;
    uint width = App->window->GetWidth() * zoom;
    uint height = App->window->GetHeight() * zoom;

    if (transform.position.x < camera_pos.x + width &&
        transform.position.y < camera_pos.y + height &&
        transform.position.x + transform.size.x * transform.scale.x > camera_pos.x &&
        transform.position.y + transform.size.y * transform.scale.y > camera_pos.y)
        return true;
    else
        return false;
}
```

F 5.5.1.1 - IsInsideCamera function

Inside the debug panel of the application we can see the number of quads being drawn. The following images show that when it is not applied every entity is drawn, in this case 30 asteroids and 1 background. But when it is applied, only the entities inside the camera limits will be drawn.

F 5.5.1.2 - Frustum Culling not applied



F 5.5.1.3 - Frustum Culling applied

5.5.2 Batch Rendering[W21]

Batch Rendering is an optimization technique used to lower the amount of draw calls sent to the GPU.

Engines have to send a set of instructions to the GPU to tell the GPU what and where to draw. These instructions are sent using common instructions called APIs, in this project the graphics API is OpenGL.

Different APIs incur different costs when drawing objects. OpenGL handles a lot of work for the user in the GPU driver at the cost of more expensive draw calls. As a result, applications can often be sped up by reducing the number of draw calls.

In 2D, we need to tell the GPU to render a series of primitives (rectangles, lines, polygons etc). The most obvious technique is to tell the GPU to render one primitive at a time.

While this is conceptually simple from the engine side, GPUs operate very slowly when used in this manner. GPUs work much more efficiently if you tell them to draw a number of similar primitives all in one draw call, which we will call a "batch".

The way it was implemented in the project was by making these batches an array of vertex data containing the information of every quad. Since all of the entities that can be created have a rectangle primitive we can pack them together in a batch to draw them in a single draw call.

We keep track of the number of indices of the batch and also a pointer to the last position of the array, to know where to add more data.

```
// Quad
uint quadVAO = 0;
uint quadVBO = 0;
uint quadIBO = 0;

uint index_count = 0;
Vertex* quad_buffer = nullptr;
Vertex* quad_buffer_ptr = nullptr;
```

F 5.5.2.1 - Batches data

The vertex data layout is shown in the figure below, it contains the model matrix of the entity, its position, color, texture coordinates and texture index.

```
struct Vertex {
    glm::mat4 model;
    glm::vec3 position;
    glm::vec4 color;
    glm::vec2 tex_coords;
    int tex_index;
};
```

F 5.5.2.2 - Vertex data

When the Render System sends data to draw an entity it is added to the current batch as shown in the following figure.

```
constexpr glm::vec2 texCoords[] = {
    { 0.0f, 1.0f },
    { 1.0f, 0.0f },
    { 0.0f, 0.0f },
    { 1.0f, 1.0f }
};

constexpr glm::vec3 positions[] = {
    { 0.0f, 1.0f, 0.0f },
    { 1.0f, 0.0f, 0.0f },
    { 0.0f, 0.0f, 0.0f },
    { 1.0f, 1.0f, 0.0f }
};

// Fill buffer (4 vertices)
for (size_t i = 0; i < 4; i++)
{
    quad_buffer_ptr->model = transform;
    quad_buffer_ptr->position = positions[i];
    quad_buffer_ptr->color = color;
    quad_buffer_ptr->tex_coords = texCoords[i];
    quad_buffer_ptr->tex_index = tex_index;
    quad_buffer_ptr++;
}

index_count += 6;
```

F 5.5.2.3 - Adding an entity to the batch

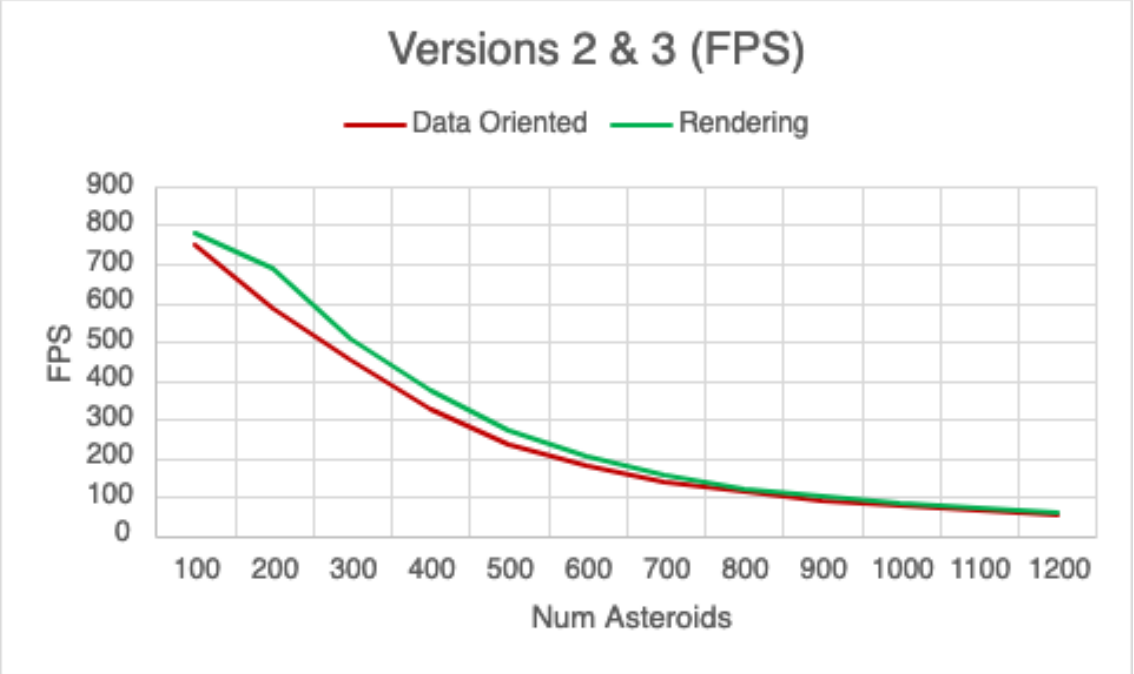
The way in which batches are processed is divided into three functions:

- **BeginBatch**: it is called every frame before the data is sent to the renderer, it resets the vertex array buffer to start a new batch.
- **EndBatch**: during the frame's update function the batch is filled with data and, when the frame ends, the function EndBatch sends the data to the GPU.
- **RenderBatch**: it performs the draw call.

5.5.3 Profiling

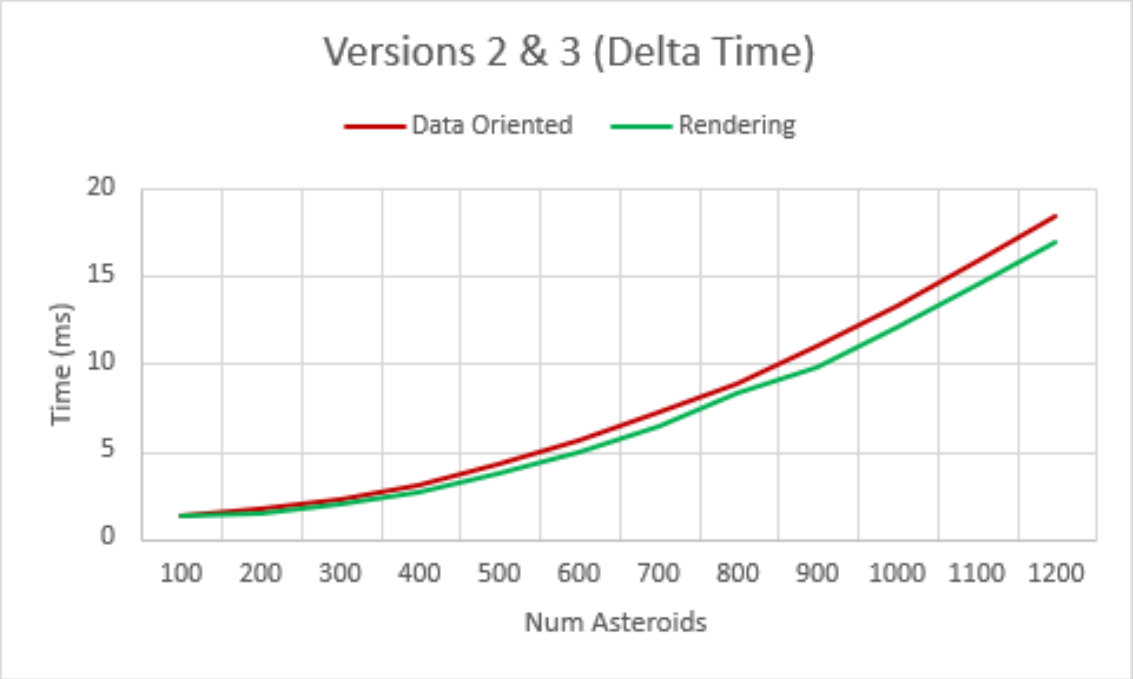
The tests performed after finishing this version followed the same methodology as the previous versions.

The following graph shows the difference in performance between this version and the previous one. There isn't much difference because, as with the previous versions, the physics calculations are what takes the most time to compute.



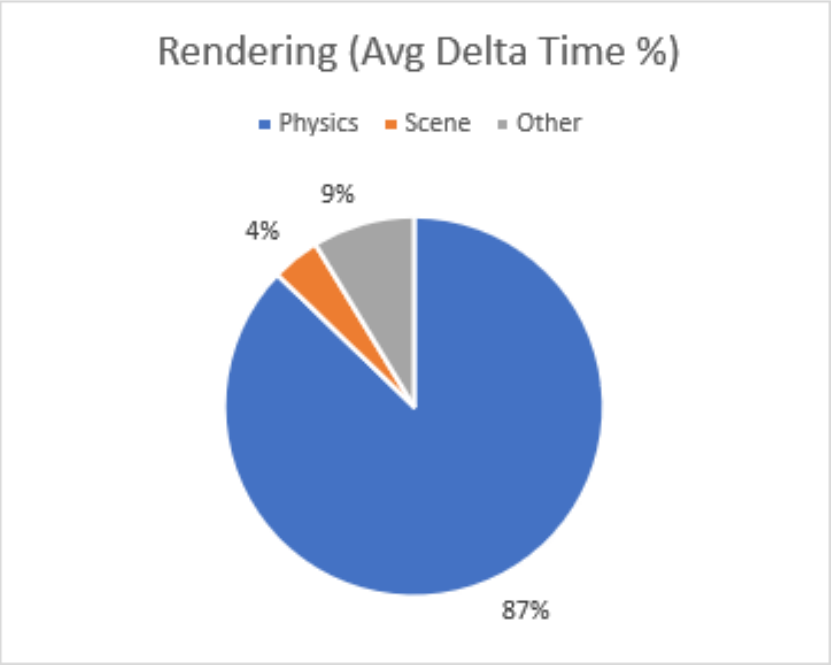
F 5.5.3.1- Versions 2 & 3 (FPS) graph

The next graph shows the difference between delta times, we can see that as the number of asteroids begin to grow, the new version outperforms the previous one and, the higher the number of asteroids, the difference between them becomes bigger.



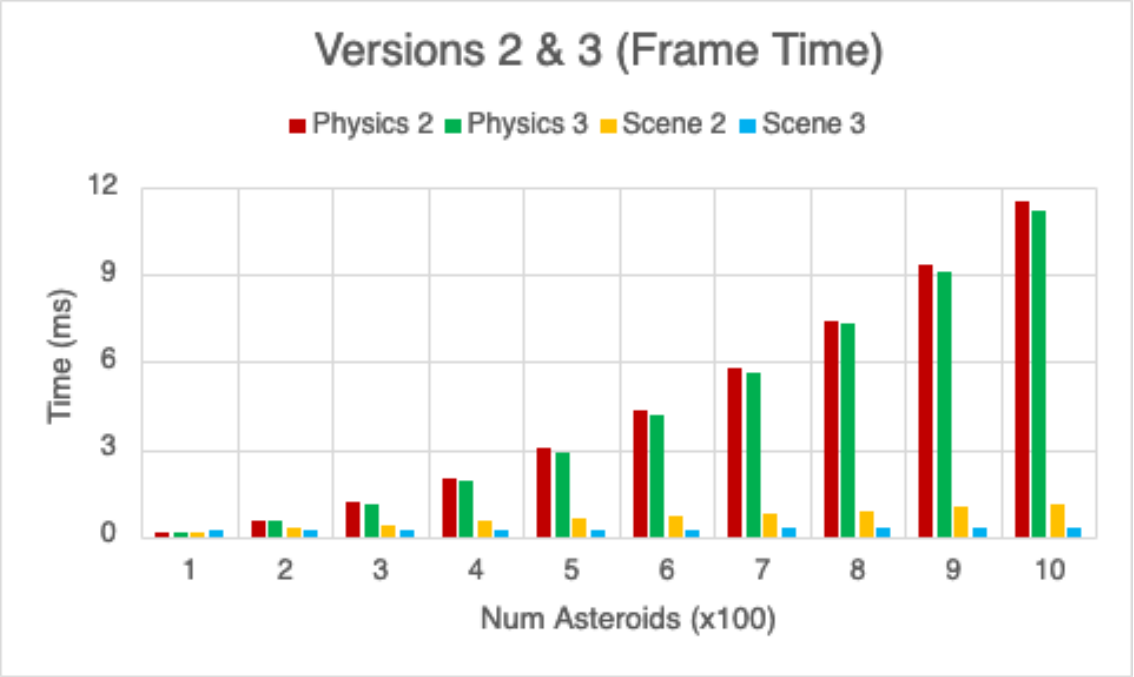
F 5.5.3.2 - Versions 2 & 3 (Delta Time) graph

The following graph reflects the percentages of the average delta time taken in a frame. We can see that the percentages are exactly the same as in the data-oriented version, but as the previous graph shows, it takes less time to calculate each frame.



F 5.5.3.3 - Rendering Optimizations (Average Delta Time %) graph

The following graph shows the comparison between both versions and the time it takes each part of the frame. We can see that the physics calculations remain mostly the same with a slight improvement in the new version. The scene operations are where the optimization was implemented and we can see that it has a positive result in the time taken to compute.



F 5.5.3.4 - Versions 2 & 3 (Frame Time) graph

5.6 Version 4 - Space Partitioning

In this section we will analyze the fourth and last version of the project. The main optimization of this version is focused on improving the physics calculations with the implementation of a space partitioning algorithm.

5.6.1 Space Partitioning

As we have already seen in the graphs of the previous versions, the physics system is the most time consuming part of the application. It takes so much time to compute because we are checking if an entity is colliding against every other entity, which results in a computational cost of O(N^2).

Space partitioning algorithms are one way of solving this problem, it consists of the process of dividing a space into non-overlapping regions and any point in the space can lie in exactly one of the regions.

By dividing the space we can now check for collisions only between entities that lie in the same subspace, greatly reducing the amount of checks we have to do.

5.6.2 Fixed Resolution Grid[W23]

For this project I decided to implement a fixed-resolution grid to subdivide the space. Each division is of the same size and we will call these divisions “cells”.

I decided to implement a fixed-resolution grid because every entity is moving and they can be fast so every frame each cell will have different entities than in the previous frame. In order to keep all the data updated, the easiest and fastest way to do so is by clearing the data and re-inserting all the entities in the corresponding cells. By having a simple data structure, it is easier to perform these calculations.

Inside the project, the grid is declared as shown in the image below.

```
uint numRows = 0;
uint numColumns = 0;

glm::vec2 pos = glm::vec2(0.0f);
glm::vec2 size = glm::vec2(0.0f);
float cellSize = 0.0f;

// Grid Cells
std::vector<uint32_t> cell_heads; //index of first item in cell inside items_list
std::vector<uint32_t> cell_sizes; //number of items inside each cell

// Items
std::vector<uint32_t> items_list;
```

F 5.6.2.1 - Grid data

We keep track of the number of rows and columns the grid has, its position and total size, and the cell size.

We have three arrays:

- **Items List:** is the main array, it holds the data of every item inside the grid. In this case every entity inside the grid is just its ID number.
- **Cell Heads:** is an array with the index of the first item inside the main array for each cell. Its size is the number of cells.
- **Cell Sizes:** is an array the number of items each cell contains. Its size is the same as cell_heads and they are aligned.

The indices inside the cell_heads array are set by calculating the number of items inside each cell with the help of the cell_sizes array.

The following image shows the function used to recalculate the grid.

```
// Clear Current Grid
Clear();

// First Pass (set number of items inside each cell)
uint32_t num_items = 0;
for (uint32_t i = 4; i < num_elements; ++i)
{
    for (uint32_t index : GetCells(positions[i], sizes[i]))
    {
        cell_sizes[index]++;
        num_items++;
    }
}
items_list.resize(num_items); // Resize items_list to actual size (but empty)

// Second Pass (set first item index of each cell)
size_t numCells = (size_t)numRows * numColumns;
for (uint32_t i = 1; i < numCells; ++i) // we start from second cell because the first cell will always start at head 0
{
    cell_heads[i] = cell_heads[i - 1] + cell_sizes[i - 1];
}

// Third Pass (insert items inside items_list)
std::vector<uint32_t> tmp_cellSizes(numCells, (uint32_t)0);
for (uint32_t i = 4; i < num_elements; ++i)
{
    for (uint32_t index : GetCells(positions[i], sizes[i]))
    {
        uint32_t item_index = cell_heads[index] + tmp_cellSizes[index];
        items_list[item_index] = i;

        tmp_cellSizes[index]++;
    }
}
```

F 5.6.2.2 - RecalculateGrid function

The first step is to clear the current grid. Then we get the number of items inside each cell.

In order to get the current cell or cells that contain each item we check in which cell lies each one of the four corners of the item. With this method we ensure that every item can be in a maximum of four cells at a time, this way we know that the item's data won't be copied a lot, but this method won't work correctly if an item is bigger than the cells' size.

After this we can now calculate the cell_heads array with the correct indices to the items_list array.

Finally we insert all the items inside the items_list array in the order of the cells.

We now have a sorted array of data containing all the items inside the grid, if we wanted to check if a specific entity is colliding we just have to get the entities that are in the same cells as the specific entity.

The following image shows the function that does this, we pass it the position and size of the entity to check and return an array filled with the entities contained in the same cells.

```
std::vector<uint32_t> FixedGrid::GetCandidates(const glm::vec2& pos, const glm::vec2& size) const
{
    std::vector<uint32_t> candidates = {};

    for (const uint32_t& cell_index : GetCells(pos, size))
    {
        // Add items inside cell to candidates list
        const std::vector<uint32_t>::const_iterator& start = items_list.begin() + cell_heads[cell_index];
        candidates.insert(candidates.end(), start, start + cell_sizes[cell_index]);

        // Check Borders
        const int cell_x = cell_index >= numColumns ? cell_index % numColumns : cell_index;
        const int cell_y = cell_index / numColumns;

        if      (cell_y == 0)               candidates.push_back(0); // Top
        else if (cell_y == numRows - 1)     candidates.push_back(1); // Bottom
        if      (cell_x == 0)               candidates.push_back(2); // Left
        else if (cell_x == numColumns - 1)  candidates.push_back(3); // Right
    }

    return candidates;
}
```

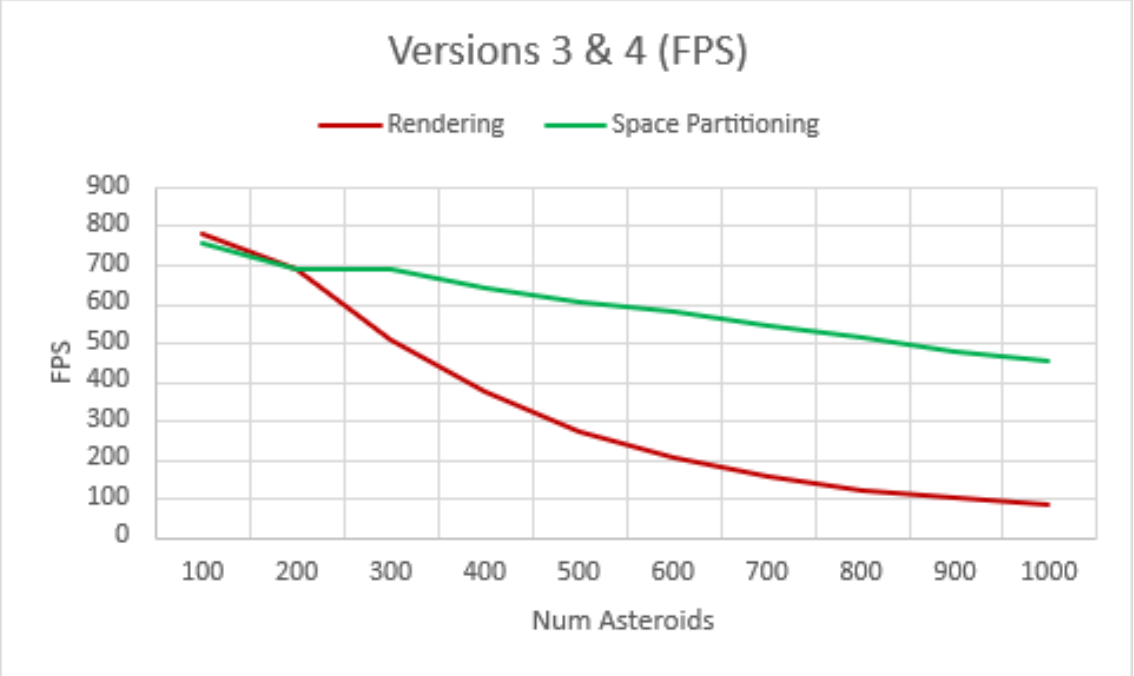
F 5.6.2.3 - Get Collision Candidates function

As mentioned before, the method used for calculating the cells that contain each item does not work with items bigger than the cell size, the borders of the game-world are bigger than the cell size. The solution to this problem is by checking if a cell is on the edge of the grid and, if it is, we add the corresponding border to the candidates list.

5.6.3 Profiling

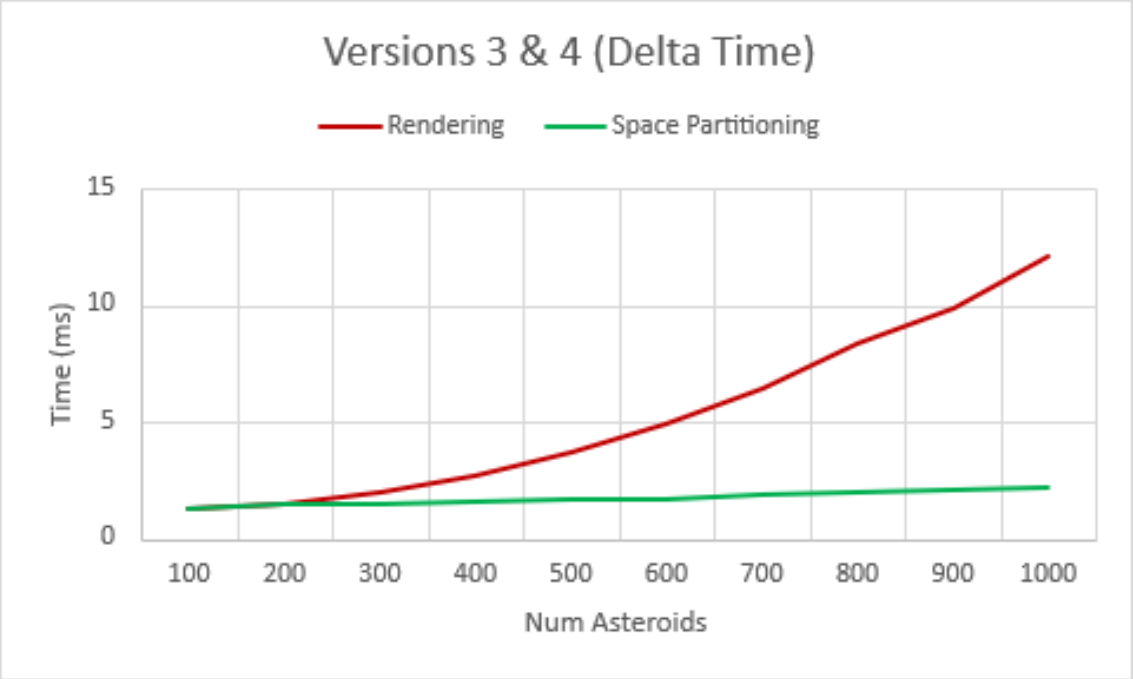
The tests performed after finishing this version followed the same methodology as the previous versions.

The following graph shows the difference in performance between this version and the previous one. We can see that at the beginning the new version is slower than the previous one, this is because of the grid calculations. But as the number of asteroids become higher we can see a clear improvement in frame rate.



F 5.6.3.1 - Versions 3 & 4 (FPS) graph

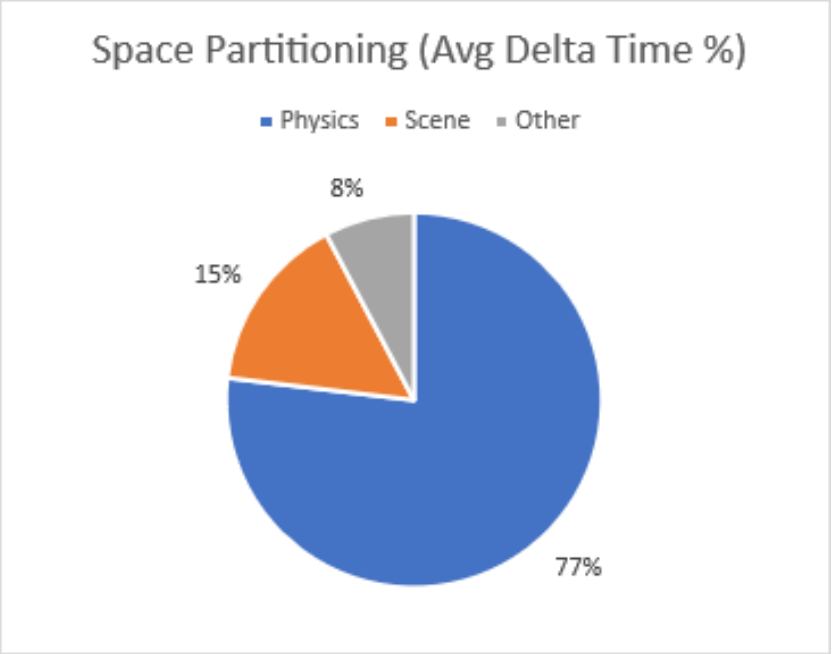
The next graph shows the difference between delta times. We can see that, from the start, the new version outperforms the previous one and its delta time doesn't grow much even at the previous version's limit.



F 5.6.3.2 - Versions 3 & 4 (Delta Time) graph

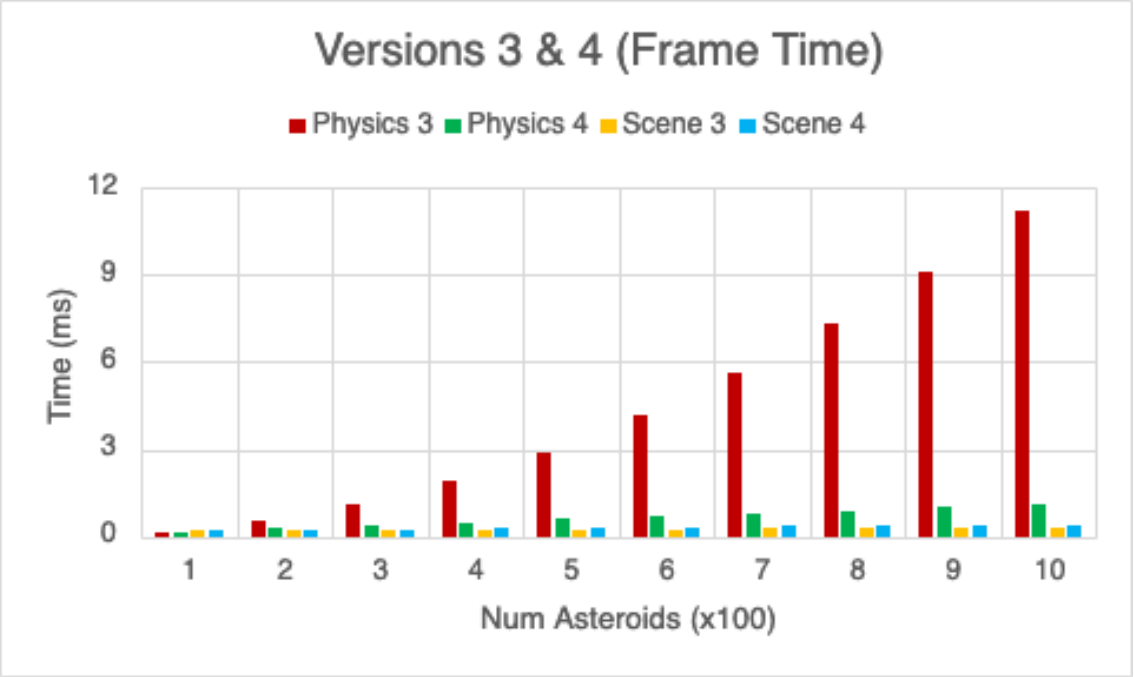
The following graph shows the percentages of the average delta time taken in a frame. We can see that the percentages have changed from the previous version: physics calculations take 10% less from the total frame's time. On the other hand, this extra time is taken up by the scene operations which now take 15% of the total time instead of the previous 4%.

This means that the time to perform all the physics calculations has been reduced so that proportionally the scene operations appear to take more time, but the time needed to perform the scene operations is exactly the same as before.



F 5.6.3.3 - Space Partitioning (Average Delta Time %) graph

The following graph shows the comparison between both versions and the time it takes each part of the frame. Scene operations remain mostly the same, but where it really changes is in the physics calculations. We can see that, with the use of the space partitioning technique, the time taken to perform the operations is significantly reduced.

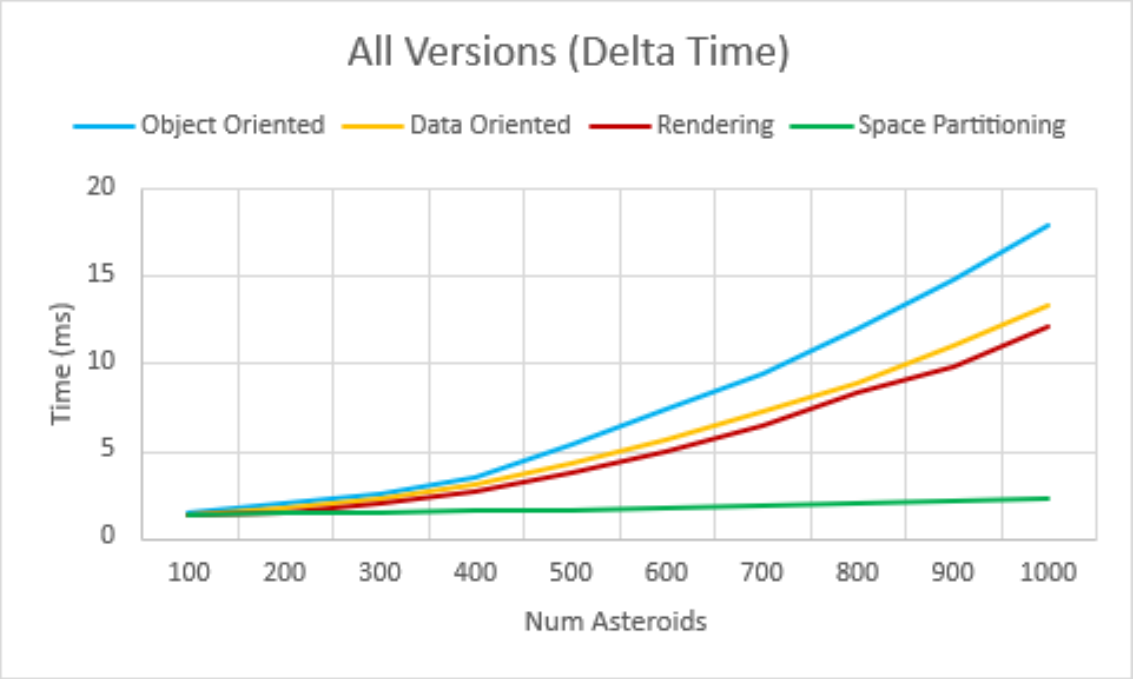


F 5.6.3.4 - Versions 3 & 4 (Frame Time) graph

6. Conclusion

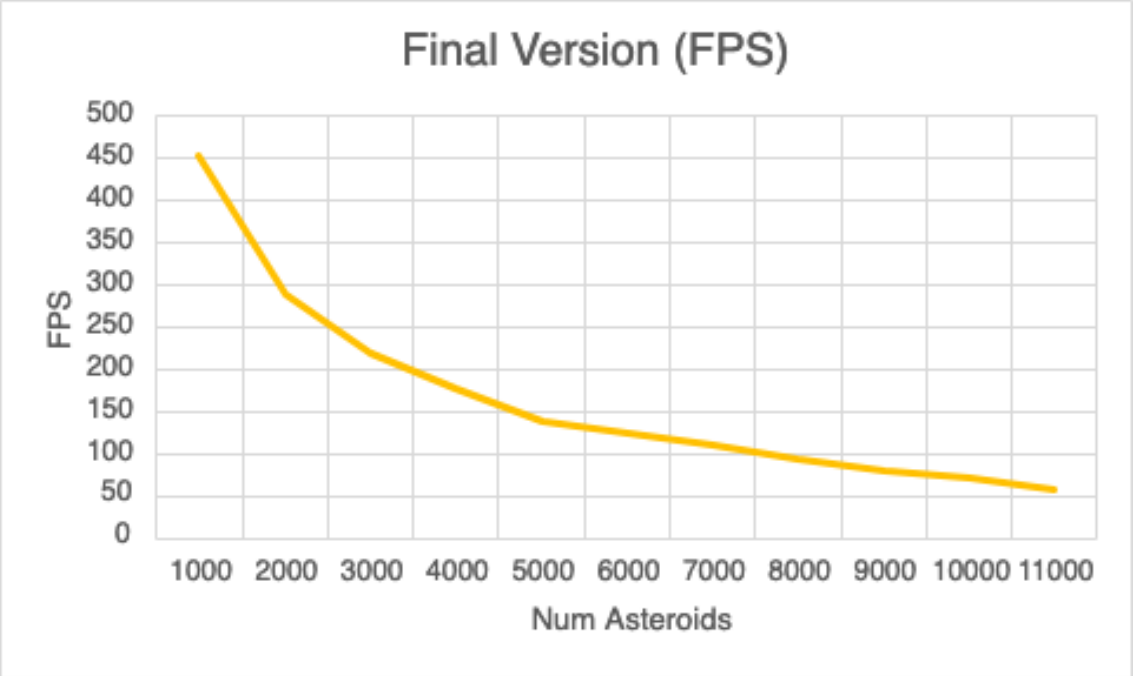
After implementing all the optimizations we have a performant application that is able to simulate many objects that interact with each other in real time.

The following graph shows a comparison between all the versions and the time it takes each frame. We see that the final version greatly outperforms the others.



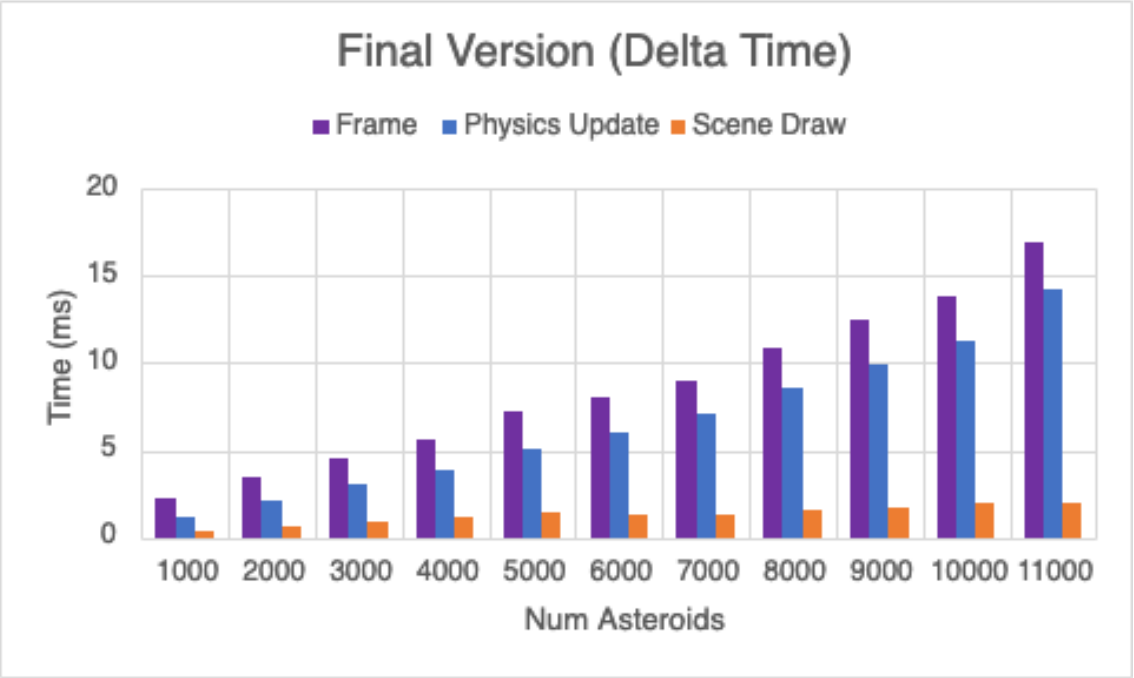
F 6.1 - All Versions (Delta Time) graph

The frame rate is so much better that we can start to have more asteroids at an acceptable frame rate, the following graph shows the FPS from 1.000 asteroids to 11.000.



F 6.2 - Final Version (FPS) graph

Finally, the graph below shows the time it takes each part of the frame. We can see that physics calculations are still the bottle-neck of the application but, even with 11.000 entities in the scene, the milliseconds needed to perform the operations is below 15ms, lower than the time taken in the first version with only 1.000 entities.



F 6.3 - Final Version (Delta Time) graph

If we go back to the general objectives set at the beginning of the development, we can analyze its state and progress overall.

- **2D rendering from scratch:** this objective has been achieved, we can render images, rectangles and lines, and there is a render pipeline that performs the necessary operations to draw everything with the use of OpenGL.
- **Applying optimization techniques that improve the performance:** this objective has also been achieved, after implementing every version the final result is able to handle more than 10.000 entities that interact with each other at a frame rate higher than 60FPS.
- **Documenting the process and creating a demo:** this objective has been mostly achieved, the only thing that has not been fully completed is the demo. With the changes in planification I decided to not make a controllable character and a small game to play but instead have a basic application with a debug panel to tweak some parameters and add or remove asteroids to the scene.

Even though the project is finished and the result is positive, there are still many things that could be improved to make it still more performant. Some of this further optimizations could be implementing a better space partitioning algorithm like a hierarchical grid or a loose double grid[(w24)], making the application run in multiple threads to distribute the workload, or performing the physics calculations not every frame but once every 5, 10, or a number of frames to not have to do costly operations so frequently.

The result of this project, Optimization techniques for a 2D engine, is public under the MIT License in the Github repository "2D Renderer", the initial name of the project.

7. Webgraphy

W 1 - Daniel Shaya. (August 5th, 2015). *The difference between Efficiency and Performance – It's not all about the Big O.* Java Code Geeks. Article, URL. The difference between Efficiency and Performance - It's not all about the Big O - Java Code Geeks - 2022

W 2 - Kevin Krewell. (2009). *What's the Difference Between a CPU and a GPU?*. Webpage.CPU vs GPU? What's the Difference? Which Is Better? | NVIDIA Blog

W 3 - Gabriel Torres (September 12th, 2007). *How The Cache Memory Works*. Hardware secrets. Webpage, URL. How The Cache Memory Works - Hardware Secrets

W 4 - DOTS packages. Unity. Webpage, URL. https://unity.com/dots/packages

W 5 - Nanite Virtualized Geometry. Unreal Engine. Webpage,URL. https://docs.unrealengine.com/5.0/en-US/nanite-virtualized-geometry-in-unreal-engine/

W 6 - Gogotengine. Main page. Webpage, URL. https://godotengine.org/

W 7 - Agile methodology: Feature Driven Development. Main page. Webpage, URL. http://agilemodeling.com/essays/fdd.htm

W 8 - HacknPlan. Webpage, URL. https://hacknplan.com/

W 9 - Optick. *C++ Profiler for Games*. GitHub. Webpage, URL. https://optick.dev/

W 10 - Chua, Hock-Chuan (July, 2012). *OpenGL Tutorial: An introduction on OpenGL with 2D grafics. Programming Notes.* **An introduction on OpenGL with 2D Graphics - OpenGL Tutorial (ntu.edu.sg)**

W 11 - Shreiner, Sellers, Kessenich, Licea-Kane. (March 2013*). OpenGL Programming Guide: The Official Guide to Learning OpenGL (Red Book)*. Addison-Wesley. https://www.cs.utexas.edu/users/fussell/courses/cs354/handouts/Addiso

n.Wesley.OpenGL.Programming.Guide.8th.Edition.Mar.2013.ISBN.0321773039.pdf

W 12 - The OpenGL Extension Wrangler Library. (2017). *Latest Release: 2.1.0*. Webpage, URL. GLEW: The OpenGL Extension Wrangler Library (sourceforge.net)

W 13 - Ocornut/imgui. Bloat-free Graphical User interface for C++ with minimal dependencies. Library. Github. Webpage,URL.GitHub - ocornut/imgui: Dear ImGui: Bloat-free Graphical User interface for C++ with minimal dependencies

W 14 -PCG, A Family of Better Random Number Generators. Webpage, URL. PCG, A Family of Better Random Number Generators (pcg-random.org)

W 15 - OpenGL Mathematics (GLM). *A C++ mathematics library for graphics programming* (**GLM 0.9.9.7)** Library. Webpage, URL. OpenGL Mathematics (g-truc.net)

W 16 - Sean Barret. Nothings/stb. stb single-file public domain libraries for C/C++. Library. Github. Webpage, URL. GitHub - nothings/stb: stb single-file public domain libraries for C/C++

W 17 - S. Gilis, A., Lewis, S (July, 2021). *Object-oriented programming (OOP)*. TechTarget network. Webpage, URL. https://www.techtarget.com/searchapparchitecture/definition/object-oriented-programming-OOP

W 18 - Jonathan Mines. (March 20th, 2018*). Data-Oriented vs Object-Oriented Design.* Medium. Article, URL. https://medium.com/@jonathanmines/data-oriented-vs-object-oriented-design-50ef35a99056

W 19 - Noel. (December 4th, 2009). *Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP).* Gamesfromwithin. Webpage, URL. https://gamesfromwithin.com/data-oriented-design

W 20 - Mikhail Semenov. (January 3rd, 2015). *Fast Implementations of Sparse Sets in C++.* Code Project. Article, URL.

https://www.codeproject.com/Articles/859324/Fast-Implementations-of-Sparse-Sets-in-Cplusplus

W 21 - Godotengine. *Optimization using batching*. Webpage, URL. https://docs.godotengine.org/en/stable/tutorials/performance/batching.html#doc-batching

W 22 - Robert Nystrom. (2014*). Game Programming Patterns.* Genever benning. https://gameprogrammingpatterns.com/spatial-partition.html

W 23 - Conkerjo. (June 13th, 2009). *Spatial hashing implementation for fast 2D collisions.* The mind of conkerjo. Blog, URL. https://conkerjo.wordpress.com/2009/06/13/spatial-hashing-implementation-for-fast-2d-collisions/

W 24 - *Efficient (and well explained) implementation of a Quadtree for 2D collision detection.* (January 30th, 2017) Stackoverflow. Blog, URL. https://stackoverflow.com/questions/41946007/efficient-and-well-explained-implementation-of-a-quadtree-for-2d-collision-det

W 25 - Molly Rocket. (2021). *Handmade Hero Chat 017 - Modern x64 Architectures and the Cache*. [Video]. Youtube, URL. https://guide.handmadehero.org/chat/chat017/

W 26 - DavidTello1/2D-Renderer. Webpage, URL. https://github.com/DavidTello1/2D-Renderer