D7049E - Project Report - Group 4

Authors:

Lana Abdulla	lanabd-9@student.ltu.se
Martin Askolin	marsak-8@student.ltu.se
Marcus Asplund	marasp-9@student.ltu.se
Alexander Eklund	aleekl-8@student.ltu.se
Andreas Form	forane-9@student.ltu.se
Jonatan Stenlund	jonstn-9@student.ltu.se

Department of Computer Science, Electrical and Space Engineering
Luleå university of technology

May 19, 2023

Contents

1 Introduction

In this report, we present our virtual interactive environment created for the course D7049E. The environment is a 3D game engine written in C# with the MonoGame framework. With the engine, we implemented a top-down game where entities spawn around the player and move to their location in order to attack them. The purpose of implementing a game was to have something to test our engine against and to use for benchmarks, since a part of the course is concerned about performance and optimizations. We wanted to test parts such as the rendering, physics, collision, AI and audio systems, and measure, for example, how the number of frames per second was affected by interacting with each system. It was then interesting to see what kind of optimizations we could perform based on our findings.

In the report, we first go into further details about our specific game idea. We then talk a bit about the Entity Component System (ECS) design and how we used it to make our engine work in a data oriented way. Then, we go into details about the implementation of the engine as well as the game, in a section that can be seen as system documentation. After that, we look at some parts of the engine that were performance-critical, and how we used benchmarks to optimized them. Finally, we talk about some future work and come to some conclusions.

2 Game Idea

When designing the game lying on top of the game engine, we wanted to make something that would be simple and easy to implement in the given time span, but also something that allowed us to test and measure all parts of the engine. Our idea was top create a top-down shooter where enemies spawn in waves and attack the player in the center of the screen, similar to games like Vampire Survivors and Call of Duty zombie mode (specifically, the level Dead Ops Arcade from Call of Duty: Black Ops). The player would then be able to repel the enemies, for example, by having a gun or a melee weapon that would collide with the enemies' colliders when swung. Such a game would be simple and consist of few unique elements, but still fun and intriguing to play. The essential entities for the game to work would be a player with health, a camera looking down at the player and following them around, a weapon held by the player which could be used to repel enemies, an enemy entity programmed to move to the player's location, and a spawner responsible for spawning enemies into the scene.

The key features we defined for the game helped us realize what kind of systems we would like to have, or at least have support for, in our engine. The game is played in a 3D environment, so we would have to implement a 3D rendering system. A top-down camera that follows the player around shows the need for some kind of camera system where the cameras are easy to move around, for example, by modifying a local transform. The moveable player also highlights the need for local transforms, and furthermore, it shows that we need

an input system. The fact that we have enemies spawning and moving towards the player shows that we need some sort of simple AI system. Since we want the player to be able to interact with the enemies, we would also like a collision detection system. To achieve realistic movement with weight and drag, we might also want to include a physics system. Finally, we discussed that possibility of having background music and audio effects for different actions, which shows the need for an audio system. Details on how the actual game implementation turned out are presented in Section 4.2.

3 ECS Design

Entity Component System (ECS) design is a technique widely used in engine and game development to achieve a data-oriented approach (for an in-depth description of data-oriented design, see (Fabian, 2018)). An engine that uses this technique is, for example, the Unity Engine (*ECS concepts*, n.d.). The main idea of an ECS is to separate data from behaviour in order to gain better performance and increased flexibility. Separation comes in the form of having three different parts with different purposes, which are entities, components and systems. Entities are defined as distinct objects with unique identifiers that don't contain any data or behaviour. An example of this could be a player entity, which would be represented only by an ID, and where the ID would be used to map between the entity itself and its components. Components are datatypes containing behaviours that can change the state of the scene. Examples of components are transforms, meshes and rigidbodies. Our player entity could, for example, consist of a transform, mesh, rigidbody, collider, input and health component. Finally, systems are responsible for iterating through the component storage and updating components accordingly. A rendering system would only be responsible for handling render components, while an audio system would only be responsible for handling audio components. An introductory guide to ECS is provided by (Terra, 2023).

There are different pros and cons that come with using an ECS design, which are often set in relation to traditional Object-Oriented Programming (OOP). Some pros are that it's easy to add new components because of the decoupling, which is essential for game engines where you often want to add new types of components for specific purposes. Components are also easy to reuse since they aren't directly related to entities (compare with inheritance in OOP). Furthermore, ECS design provides better performance for systems where many objects are present at the same time. This is a result of the efficient querying used to find the components that are to be modified by a specific system. Consider a game with many different entities that contain a number of different components. Then, the rendering system would query for all render components, practically ignoring which entities they belong to, and perform computations on just these components. Some cons with an ECS design is that there are no well established best practices, which means that developers have to be careful not to create implementations that are ineffective and don't have

any performance benefits compared to OOP, especially regarding the component storage and querying.

For our ECS design, we created a first draft of an API outline (located at `https://github.com/Mehdows/D7049E-game-engine`) showing classes we would like to implement. Using the API outline, we then created an ECS diagram showing how the different parts would be connected, as shown in Figure 1 (the diagram has been updated during development to represent the actual implementation). The idea was that we would have a main game loop that would contain an update method, which in turn would call the update method of each system. As stated earlier, one of the challenges is how the systems query for relevant components efficiently. Our solution to this would be to use archetypes in order to group together entities based on their component types. This would allow us to query for a desired archetype, and the result from the query would be the components we want to modify. How this was implemented is explained in Section 4.1.

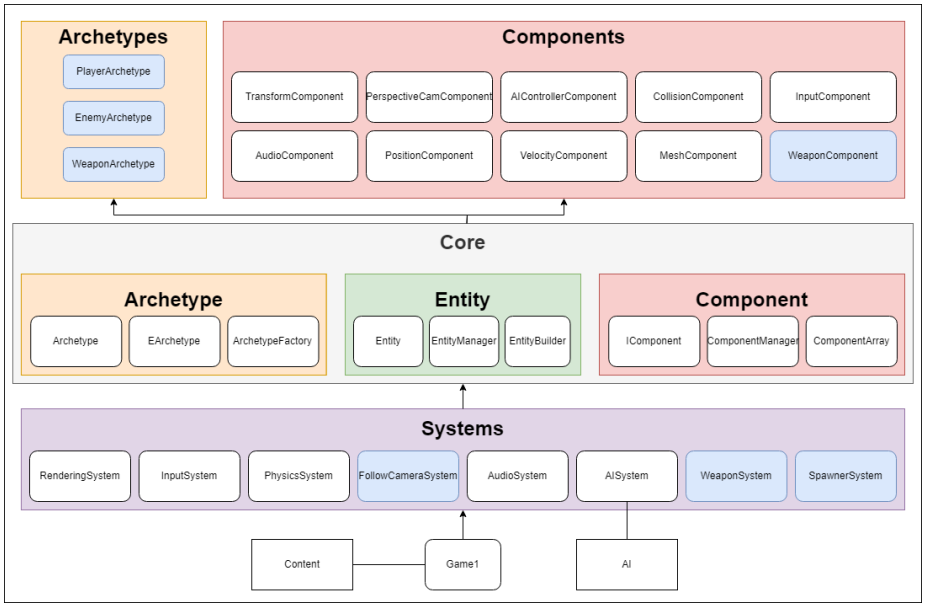

Figure 1: ECS diagram of the engine (blue boxes show game specific parts)

4 Implementation

4.1 Game Engine

The game engine section will provide a overview of the game implementation process and explain the elements that contributed to its development. As a group we identified three essential building blocks which formed our foundation

for our game engine and these were Monogame, BEPUphysics and the game loop. In the following sections we will delve deeper into each building block.

4.1.1 Monogame

The game loop inside Class Game1 in *Figure.1* is the central mechanism that drives the flow of the game. It ensures that the game's logic is updated and rendered at a consistent rate. For our engine we used four main stages: initialization, load content, update and draw. Our first and second stage were only called once on startup of the game. The initialization stage is where entities with their respective components were created. Active systems requiring entities (FollowCameraSystem) where passed their required entities and passive systems where initialized. The next stage in our game loop was the "load content" stage where we could use the Monogame content pipeline to import FBX models, sound effects, textures assets to be used for our components. The last two stages "Update" and "Draw" ran in parallel and focused on updating and drawing all systems like the input, physics and rendering systems.

4.1.2 Game loop

The game loop is responsible for controlling the flow of the game and is used for ensuring that we have a smooth and consistent gameflow. It is used as being a loop that is continuous and updating the game state as mentioned before. This action provides us with seamless flow since it will calculate all the changes in the game regularly. The changes for example refers to the character's movement, actions, physics and AI. The game loop also offers regulation of the frame rate to contribute to a seamless experience.

4.1.3 BEPUphysics

The BEPUphysics v1 physics engine provided game essential features like collision, gravity and in our case the movement system. The collision component was integral to our game's mechanics and gameplay. It allowed us to accurately determine when and where collisions occurred, enabling interactions between different objects within the game world. Whether it was a player colliding with an obstacle or two objects colliding with each other. We opted to use the collision shape on top of an entity to drive the movement of entities using their individual linear velocities. This provided a great feel as it allowed us to have accelerating, decelerating and braking movements when running opposite of the entity's momentum. As for the combat part of the game we still had more to develop as we lacked some components like health for both player and AI but instead we had knockback interaction between the players sword and AI collision shape when collision occurred.

4.2 Top-Down Shooter Game

We initiated the project by engaging in discussions about the game idea, aiming to define fundamental concepts, mechanics, and goals. This process allowed us to visualize the desired outcome and facilitated the implementation phase once our game engine was ready. An illustration of our initial idea from the first week can be found in Figure 2.

Next, we proceeded by identifying the various entities that would exist within the game, such as players, enemies, and props. Additionally, we determined the specific components that would be associated with each entity. For instance, we recognized that the player entity would require components such as Input, Transform, Velocity, Health, Mesh, Collider, and Rigidbody.

Once we established the entities and components, we designed systems responsible for processing these entities and their associated components. These systems encompassed rendering, input, physics, and AI (for enemy behavior), as shown in Figure 1. The implementation of these systems enabled interaction with our entities and facilitated their integration into the game.

Following the development of these systems, we proceeded with game testing, allowing us to identify various bugs and further enhance the gameplay experience.

What resulted was a top-down game where the player spawns in the middle, with a sword rotating around the body in a circle. Enemies spawn from the corners of the screen and target the player using the AI system. Due to time constraints, we were unable to fully implement the health system and wave mechanic into our game. The health system would have made the player take damage if they collided with enemies, and the enemies take damage if they collided with the sword. One way to "beat" the enemies now is by pushing them off the plane. The wave mechanic would have involved spawning additional enemies at specific time intervals after the demise of the last enemy. With each successive wave, the number of enemies would increase, gradually escalating the game's difficulty until the player eventually died. While this feature would have added depth and intensity to the gameplay experience, we thought as a group that the primary focus of the course centered around the game engine itself rather than gameplay. Instead, our focus was placed on the engine, its functionality and optimizations.

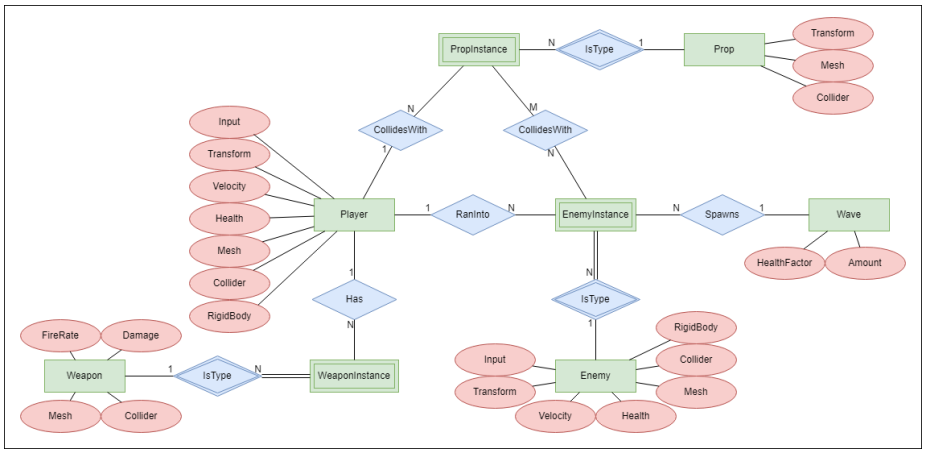

Figure 2: ER diagram showing the planned design of the game

5 Performance Critical Parts

When implementing something as complex as a game engine, it's important to measure performance and try to identify bottlenecks and performance critical parts. This can be done in multiple ways, and with a number of different tools. Perhaps the most simple and straightforward way is to play test the engine, or use some predefined benchmarks for stress testing (more on benchmarks in Section 6).

To help us identify performance issues, we implemented a frames per second (FPS) counter and a memory usage variable showing an approximation of the total amount of memory currently allocated by the .NET garbage collector for managed objects. A sudden drop in FPS may mean that something triggered some sort of bottleneck, while large amounts of memory usage may mean that something is poorly optimized. For a small game such as ours, if played normally and not stress tested, the memory usage should stay at around a couple of megabytes.

By simply playing the game, we noticed some odd behaviour regarding the collision system. The FPS counter always stayed at 60 FPS, and the memory usage was always between 5 and 15 MB, but during collisions between entities the game froze briefly for a couple of milliseconds. Furthermore, this only happened on some computers, and we suspected that it had something to do with the type of processor used. It only seemed to happen on systems using AMD Ryzen processors.

To aid us further in our investigation, we used the profiling tool AMD uProf (*AMD uProf Performance Analysis*, n.d.), which is used for analyzing applications running on AMD processors (there are similar tools for Intel processors). The tool can analyze, for example, CPU utilization, instruction execution, and

memory access patterns. You use the tool by setting up a profile, and choosing an application to run for a certain amount of time. After executing a profile, you are provided with a timeline of the events. For our collision problem, we would expect to see some sort of spike when a collision is occurring.

We ran our profilings for the application on a computer with an AMD Ryzen 5 2600 Six-Core Processor (3.85 GHz), with 16 GB RAM, running on Windows 10. The application is played normally, meaning that the player spawns in and let's the sword collide with a couple of enemies coming in from the corners. The profiling is always run for 30 seconds. Results from the first run are shown in Figure 3.

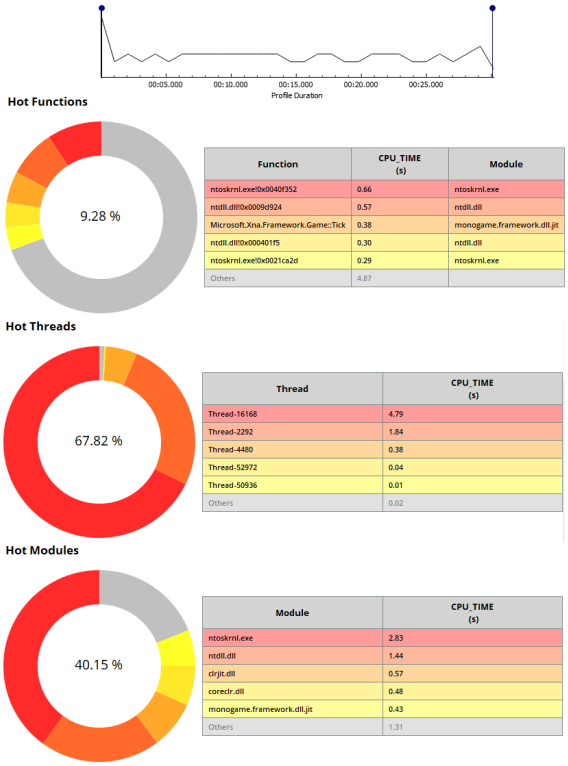

Figure 3: uProf profiling from playing normally without modifications

As we can see, the CPU activity is steady and doesn't show signs of spiked activity during collisions. The hottest module identified is ntoskrnl.exe. This module is the kernel image for the Windows operating system, and provides essential functions and services for it. This is most likely not related to the game application, and probably occurs due to other Windows system problems. It could, for example, be hot due to starting and exiting of the application.

We tried to identify Thread-16168 without luck, but it's safe to assume that this thread is related to the Windows kernel. In total, it seems as though the MonoGame framework is responsible for relatively little CPU consumption, with the hottest function being Microsoft.Xna.Framework.Game:Tick which is responsible for updating game logic in the game loop.

Although the profiling didn't aid us in identifying the strange collision problems, we soon identified the problem cause to be Visual Studio console logs. There were lines of code that wrote console lines during each frame where objects were colliding. For Visual Studio users, these console lines made the game freeze briefly. Removing them made the game run smoothly. It's worth mentioning that a second profiling was run with the console logs removed, but they didn't provide any different results from the profiling with the console logs included. The Visual Studio console logs were only an issue for AMD Ryzen processors and not for Intel processors. No other processors were tested.

A more conceptual and hard to prove bottleneck is the component querying. As we know, this part is hard to get perfect, and it's easy to implement in a way that leads to some performance issues. The best ways to identify problems with the component querying is to look through the code for errors, and also to use benchmarks that spawn a lot of entities holding components that are queried for every frame. Under normal playing circumstances, our profiling in Figure 3 doesn't show any symptoms for problems with the querying.

Other relevant potential bottlenecks to look out for are physics management, resource loading, audio processing and input handling. Through our testing, we didn't notice any particular abnormalities with these parts. There may also be problems with the rendering pipeline, but since this is not a course focused on 3D graphics, we have chosen to not put much focus on this.

6 Benchmarks and Optimization

Benchmarking are techniques used to measure the performance systems or chunks of code. It usually involves running standardized tests that utilize all or certain parts of a system, while measuring performance such as frame rate or memory usage. Benchmarks are good for identifying bottlenecks, and they also provide clear procedures that can be tested against to ensure that your optimizations have yielded better results than before. As presented earlier, our game engine has implemented an FPS counter and a memory usage variable, but we also needed to implement some sort of benchmarking scenario where we tested the features of our engine.

Since our engine makes use of the ECS design, while also implementing a physics and collider system, we wanted to make sure that our engine could handle large amounts of component querying, where physics and collider components were modified each frame. We defined a scenario where the player spawned in as usual, while enemies spawned from each corner of the screen every 0.2 seconds. The simulation lasted for 30 seconds until exiting by itself, resulting in a total of 600 enemies spawning. Note that this amount of ene-

mies existing at once would not be realistic during normal gameplay, since they almost don't even fit on the screen.

The benchmark we defined simultaneously tests the component querying, physics, colliders, 3D rendering and AI implementation, which makes it suitable for evaluating the overall performance of the engine. Results from running the simulation shows that the memory usage steadily rises to around 50 MB, which is to be expected. However, the FPS counter shows a steady 60 FPS even though the simulation gets visibly slower at the end. This is most likely due to the Update method being called at a steady rate, while the draw calls are taking longer than expected. Another cause may be an incorrectly implemented FPS counter, but we have tried some different implementation techniques that all yielded the same results.

We also ran the AMD uProf profiler on the benchmark, from which the results are shown in Figure 4. As we can see, the CPU is experiencing an overall higher load compared to normal game execution (Figure 3), and now the BEPUphysics module also makes it's entry to the list of hot modules. BEPUphysics.Constraints.SolverGroups.SolverGroup::SolveUpdateable is the most prominent function except for the Windows kernel. This function is responsible for resolving constraints and collision between objects in the physics simulation, so it makes sense that it's working hard when there are many physics objects in the scene that move around and collide with each other. BEPUphysics.Constraints.Solver::UnsafeSolveIteration is a function that's called in SolveUpdateable for each constraint, applying physics to objects. The GetComponent function that we defined for the component array is also identified as a moderately hot function, which is the result of the many hundred components being fetched every frame.

From seeing these results alone it's hard to say if the physics system or component array needs to be optimized in any way, since there is no absolute standard that works for all types of applications. However, it's always important to monitor performance and try to make improvements where they seem possible, while also looking out for changes that create abnormalities in the performance.

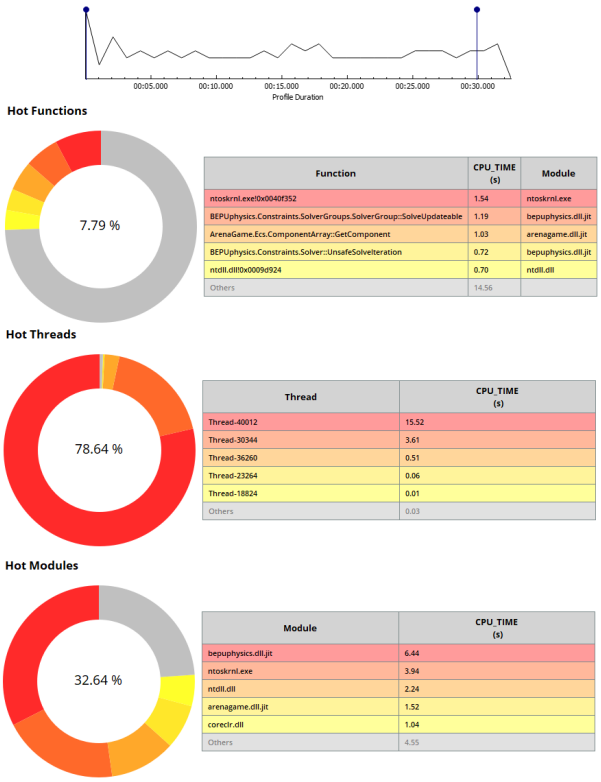

Figure 4: uProf profiling for the benchmark

For us, optimizations were seldom based on benchmarks and profiling results, with the exception being removing the console logs for optimizing the collision system. Instead, we mostly performed optimizations as part of the iterative design and coding process. The component querying, as an example, was first planned to be smaller and simpler by only consisting of an entity manager holding a component array, querying based on predefined archetypes. Not due to performance issues, but due to the code not being flexible enough for our purposes, we redid the entity and component system until it felt easy and flexible to work with. This resulted in the core that's shown in Figure 1.

Another example of optimization based on flexibility is the focus we put on iteratively cleaning up the code, by which we mean removing old code, simplifying loops, removing code repetitions and making reusable helper functions. While the main purpose of this was to make it easier for the developers to work with the code, it also increases engine performance. Less code that is more efficient means that the engine will run more smoothly as well.

A neat feature of the MonoGame framework worth mentioning is that the resources, such as sprites and 3D models, are loaded once during the initial-

ization phase of the game. Resource loading is done via the content manager, and it can only be done in the LoadContent function which is called by the MonoGame framework. Resource loading can be a heavy operation, and doing it at the start of the game means that the game might take a bit longer to start, but no further loading is needed during the rest of the run-time.

Due to us not finding any major performance issues during or testing and benchmarking, we didn't pursue any other optimizations than the ones we presented here. If we had more time, we could have looked at some other optimizations that are discussed briefly in the next section about future works.

7 Future Works

After implementing the rudimentary systems for the game engine we've got to consider the future steps to take to achieve our game idea. One thing to make it more approachable to the player is to add an User Interface (UI) system. A UI system will enable us to create menus, buttons, and other interactive elements which could be used not only as a menu for the game but also in game for different features. This will make the game more user-friendly and improve their experience.

Additionally, adding a networking system to enable multiplayer functionality could generalize our game engine and make it easier for us to add multiplayer in the future. This will allow players to play together and compete against each other. Perhaps allowing a high score tab where each player could see their ranking.

One of the main parts of our game, the combat system, is quite simple at the moment. The current combat system lacks depth and results in repetitive and boring fights. To improve this we could introduce a wider range of combat mechanics including different weapons, abilities and other strategic elements that would allow the player to think more tactically. By introducing critical hits, combos or special moves the system would be more interesting and reward the player in fights.

Similarly, the lack of diverse monsters can also make combat boring. Adding more diverse monster both graphically and with different abilities, stats and difficulty. This would allow the player identify hard monsters to avoid or to risk fighting for better rewards. It would also introduce more strategy to fights, where learning different monsters' skill sets would give you an advantage when fighting them, since you could predict their moves and have strategies to counter them.

As for the engine, you could introduce further optimizations by, for example, placing systems in their own threads to achieve concurrency. To reduce cross-referencing between systems, you can also instead send event-driven messages between the different parts. Furthermore, you could expand on the UI aspects by creating a simple and standardized interface for adding new entities and components. This would allow developers to make changes and use the engine without having to deep dive into the code and understand all core components.

8 Conclusion

In this report, we presented our project, how we planned it, and how we implemented it. We also showed our implementation of a top-down game created in the engine, which purpose was to have something to showcase and test performance against, and a benchmark which aimed to stress test all parts of the engine. By play testing, using the benchmark, and profiling with AMD uProf, we identified some performance critical parts that could be used as a basis for optimizations. These parts were the collision system, and also somewhat the BEPUphysics constrains update and ComponentArray.GetComponent methods.

Optimizations were performed, where the main one was to remove console logs during collisions. Other optimizations were done as a part of the iterative development process, driven by a desire for better code. They mainly consisted of cleaning up, and making code and entity-component handling more effective. We also discussed further optimizations that could be considered, such as event-driven messaging and threading.

When working with this project, we realized the importance of planning when you're intending to create an ECS, and also the importance of often measuring the performance of your implementation. The lack of clear universal standards means that there are many ways to implement an ECS design, and careful considerations have to be made in order to ensure that the handling and querying of components is efficient. Game engines are sensitive to performance since they are controlled and experienced by humans in real time, so good performance has to be expected at all times. Benchmarks are useful for having a standardized way to test the limits your engine, and profiling tools such as AMD uProf are good for identifying potential hot functions.

We hope that this report has given clear insight into how we planned and implemented our work, as well as showing some insight into developing game engines in C# with the MonoGame framework.

References

Amd uprof performance analysis. (n.d.). Advanced Micro Devises. Retrieved 2023-05-16, from https://www.amd.com/en/developer/uprof/uprof-performance-analysis.html

Ecs concepts. (n.d.). Unity Technologies. Retrieved 2023-05-09, from https://docs.unity3d.com/Packages/com.unity.entities@0.1/manual/ecs_core.html

Fabian, R. (2018, October 8). *Data-oriented design.* Retrieved 2023-05-09, from https://www.dataorienteddesign.com/dodbook/

Terra, J. (2023, March 1). *Entity component system: An introductory guide.* Retrieved 2023-05-09, from https://www.simplilearn.com/entity-component-system-introductory-guide-article