A Scenario-Based evaluation of Game Architecture

Frank Hvidbjerg Hansen
Hadi AL Halbouni

July, 2020

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona Sweden

Contact Information:
Author(s):
Frank Hvidbjerg Hansen
E-mail: frankhvidbjerg@gmail.com

Hadi AL Halbouni
E-mail: hadiblade09@yahoo.com

University advisor:
Fabian Fagerholm

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

When developers or organizations need to develop a game, simulation or a similar project, they phase the question of whether or not to use a game engine as well as the question on which one to use. Are all game engines the same or does the architecture change and how is the game design different between various game engines? The objective of this thesis is to research these questions as well as giving a concrete understanding of the impact of picking one engine over the other and how each engine influences the way games are developed and answer some more specific questions regarding architecture and usability.

A project was designed with the goal of developing a game. This game was developed by two separate teams over a period of 6 weeks, using two different game engines. The development was split into separate iterations done simultaneously between the teams and questionnaires were filled in to gather data. The game engines used for projects had similarities but also things which were different. Each engine offered ways to speed up development by allowing the developer to reuse and distribute changes among objects to reduce work. The differences caused one engine's code architecture to be more complex than the other while allowing a better code structure as well as adding more time to learn how the engine handles certain things such as collisions.

In conclusion, there is an importance to properly evaluating different game engines depending on the project a developer or organization is creating, not evaluating this properly will impact development speed and project complexity. Even though each engine has their differences, there is no superior game engine as it all depends on the project being developed. The game developed for this project was only touching on certain areas related to 2D games.

Table of content

Table of content 3

1. Introduction 5

1.1. Aim 5

1.2. Motivation 6

1.3. Scope 6

2. Related work 8

3. Research Method 10

3.1. Research Questions 10

3.2. Software Architecture Analysis Method 11

3.3. The Project 11

3.4. Iterations 12

3.5. Method of gathering data 14

3.6. Final Interview 15

3.7. Prior knowledge 16

4. Analysis and Results 17

4.1. The Project 17

4.1.1. Iteration 1 17

4.1.2. Iteration 2 17

4.1.3. Iteration 3 18

4.1.4. Iteration 4 18

4.1.5. Iteration 5 20

4.1.6. Iteration 6 21

4.2. Components used in the project 22

4.2.1. Components used in Godot [7] 22

4.2.2. Components used in Unity [8] 23

4.2.3. Comparison between components 24

5. Discussion 26

5.1 RQ1. How is game architecture affected by picking one engine over another? 26

5.2 RQ2: How does choosing a game engine affect the structure of the overall game code? 27

5.3 RQ3: To what extent can game engine components in each game engine be reused? 27

5.4 Validity Threats 28

5.4.1 The Game Project 28

5.4.2 Oversights in Interview questions 28

5.4.3 Team size and game engines 28

5.4.4. Meeting issues 29

5.4.5. Using Ourselves in the experiment 29

6. Conclusion **30**
6.1. Future Work 30

7. References **31**

8. Appendix **33**
8.1. Notes taken during development from the Godot team 33
8.1.1. Iteration 1 33
8.1.2. Iteration 2 33
8.1.3. Iteration 3 34
8.1.4. Iteration 4 35
8.1.5. Iteration 5 35
8.1.6. Iteration 6 35
8.2. Notes taken during development from the Unity team 36
8.2.1. Iteration 1 36
8.2.2. Iteration 2 36
8.2.3. Iteration 3 36
8.2.4. Iteration 4 37
8.2.5. Iteration 5 37
8.2.6. Iteration 6 38
8.3. Further list of components 39
8.3.1. Godot Components 39
8.3.2. Unity Components 39

1. Introduction

If a developer has to work on or create a game, they have a few options. Either they learn how to do everything from scratch [17], such as graphics, physics, networking, user-input, collision handling among the common aspects of a game, which in turn is the same as creating their own game engine which also requires a solid understanding of math and algorithms [12]. This path can add a lot of overhead and cost to the project and might end up delaying the game or project significantly and is not recommended unless there are good reasons to do so [2], such as wanting to learn the process behind making a game engine or being a large enough team where having a custom game engine made specifically for a project is a better option [1].

Another option would be to pick a pre-existing game engine [3] where other experienced developers have written all these components and functions for you, and made them easy to use. This can contribute to decreasing production costs and helps developers meet project deadlines on time. Also depending on the engine of choice, there might be a lot more documentation that would help speed up development. The biggest concern here is picking the right engine for the task.

There are a large number of game engines available on the market, some being made to complete a certain project such as the source engine being made for Half Life 2 [22] and the unreal game engine being made for the game called Unreal [23] and some like Godot and Unity being made as a general use engine. Source and Unreal have evolved over the years to also be usable for any type of game. Due to this large sample size, it is difficult to evaluate all game engines as it would simply take too long. Since most game engines are developed to complete a common goal, that is creating a game, It would be possible to get a decent understanding of game engines to be able to answer our research questions by just comparing a smaller sample.

1.1. Aim

The aim of the thesis was to research how a game engine’s architecture affects the software engineering aspects of games, this was done by using a scenario based approach. Scenario derived from latin scaena, meaning scene [19] is often used to describe a story or progress [20], in the case of this thesis, a project was used and split up into smaller tasks or “scenes”, a sort of state the project would be in each week. Additionally the scenarios were split into smaller sections or sub tasks.. The project consisted of creating a game using two separate game engines(Unity and Godot).

The main target of research was to investigate the game components, how they’re handled, how they can be used together, created and removed. During development, notes would be taken and in turn analyzed to be able to elicit answers about the architecture of the game engines. Since the tasks were small and clearly defined, it would be easier to compare notes between the engines.

Taking these notes were primarily used to help remember what was done and ensure there was enough data to base the final study on. as well as giving something to compare the different engines on. In the end of the project phase, a final interview session was held to gather some additional information.

By doing this, it was possible to analyze all of the data and perform an evaluation on the game engines design patterns [1] and architecture. as well as comparing how each game engine handled different aspects of game development such as textures, sounds, physics, collision handling as well as how each game engine uses different names and methods to make these aspects interact with each other.

1.2. Motivation

Whether making a large or small game, picking the right engine is crucial to making your workflow more efficient and the end results better. Game engines might differ in different aspects such as performance, compatibility with different platforms, names for different objects in the game, programming language, and architecture. Another thing which will be different is the way to represent certain objects, such as the player object, platforms, enemies, collectable items or other objects in a game.

A well-designed game engine will have clear definitions for each object while other engines might have vague definitions or none at all [3]. The main aspects of game engine architecture is reusability, modularity and extensibility [17]. According to Charrieras and Ivanova "A given game engine is not tied to one game but can be used for different games" [17]. The definition fits for the purpose of this thesis in the sense that even though it is described to prove that an engine can be used for several types of games, looking at the individual parts of an engine it is possible to figure out if a part can be used for several different things and whether objects can be reused.

The study for this thesis is important as it helps answer some questions related to the game engine and the difference in reusability. The purpose of this research will be to help give a better understanding on how game engines differ and what they have in common. What components are used and how they work together to make a game. With this knowledge it is possible to teach the reader how they can make their own observations on different game engines to more easily pick an engine for their project.

1.3. Scope

The scope of the project was enough to answer the questions but not enough to give the user an advanced understanding of all aspects of game engines. The project that was conducted is a 2D platform game which contains a player object which the user can move, some platforms the player can jump on, some coins the player can collect and a flag which signifies the end of the game. Game engines can do more than this project will show such as handling networking, exporting to multiple platforms, particle systems, advanced physics and rendering (ray tracing, marching cubes), path finding [7][8] and other functions that are possible to reuse in several games.

Reusability being the ability to reuse different aspects from previous implementations or parts of the software [18]. In the case of this paper, and relating to game development, it can be seen as things which can speed up the workflow for the developer and allow them to not have to redo the same thing multiple times.

This research has not covered every possible kind of reusable game engine element, but rather provide the reader with an idea of the issues they might face when picking a game engine which will help them base their own understanding of engines and help them do their own research in the event that they were to start their own project or work on a project as part of a larger team or company.

Instead of covering too large of a scope of game engines the time instead has been spent on a smaller scope to get a better understanding on how separate game engines handle the game architecture, the development progress and how different components of the game engine work together to create a game.

Two game engines which were interesting to analyze, were Godot and Unity.

Godot is a free, open-source game engine first released in 2014 [13]. It was originally developed by two people, Juan Linietsky and Ariel Manzur but have now grown into an advanced engine with over a thousand contributors. One of the main selling features of Godot, is that it is a very lightweight engine, contained in a single executable file, which requires no installation [21], meaning after downloading it, you simply run the executable and it is up and running. This game engine is free to use and requires no license to publish or has any fees attached to using it.

Unity on the other hand is a closed source game engine first released in 2004 [8] which is also free but requires licenses when companies grow in sales requiring each developer in the company to pay for such a license. It was developed by a larger team and the company behind it has over 2,000 employees. This game engine requires longer installation time and has a longer setup process.

Both engines allow the development of 2D and 3D games but this thesis will focus on the 2D aspect.

The main reason for picking these two engines is that they show a wider spectrum of game engines by comparing a small and newer engine with few developers and a large engine which has been in the market for over 15 years [9] and been used by big companies like EA Sports, Blizzard Entertainment and Activation [9] and which is developed and maintained by a large company.

2. Related Work

This chapter will discuss similar scientific research regarding areas related to game engines architecture, game development, game engine comparison, and software frameworks. This was done to get a comprehensive understanding of our research area, and trying to give some technical definitions and answer some questions that the reader might think of. In search of the software engineering scientific articles, and related research we have used Google scholar, the ACM digital library(Association For Computing Machinery) as well as regular Google search.

Lewis and Jacobson gave an early definition of game engines, they introduced it as a collection of modules of simulation code that are used by game developers but don't directly specify the game's behaviour or game's environment. In general, the engine includes modules for handling the input, the output, and generic physics/dynamics for the game world. They introduced the game engine in a hierarchically structured view, where on the top level there is the virtual world or scenarios where the user interacts with the game, the second level is the game code which handles most of the game graphics, player behaviour, game flow, parameters, and networking. The rendering engine is the core of the game which incorporates all the underneath complex code to render and identify the player's view and introduces it to the graphics driver [3].

Important questions and major points were declared by Anderson, et al. to be answered for making a precise and specific definition to game engines, like where the boundary is between game and game engine, how different genres affect the design of a game engine, how low-level issues affect the top-level design, and if there are specific design methods or architectural models that are used, or should be used, for the creation of a game engine. Answering those questions could give a better understanding of game engines and its major modules [10].

Blow, J shows that game engine development requires extra knowledge in addition to programming experience, the research suggests that the programmer should have a solid foundation in math, especially linear algebra, and geometry in 2D and 3D in order to be able to write a game engine. Besides math, the programmer should have a good understanding of a wide variety of algorithms like spatial partitioning, clustering, and intersection and clipping of geometric primitives [12].

Petridis, et al. proposed a methodology of game engine selection for High Fidelity Serious Games. This methodology uses six main metrics for comparison between different game engines, those metrics are audiovisual fidelity, functional fidelity, composability, accessibility, networking, and heterogeneity. Audiovisual fidelity includes the capabilities of a game engine for rendering, animation, and sound. Functional fidelity defines the ability of scripting support, AI technology, and physics. The composability compares between different engines in terms of import/export capability and a developer toolkit. The accessibility includes the learning curve, documentation and support, licensing, and cost. For networking there are two types of networking models: client-server and peer to peer. Heterogeneity is the ability to support multi-platforms. They used this methodology to compare four different game engines Quest3D, Blender, Unreal, and Unity. Results of their comparison show that the Unreal 3 engine outperforms all other engines in terms of audio and visual fidelity, functional fidelity, and composability, but it's high costs cause a limitation in accessibility. Quest3D outperforms the Unity engine in its networking capabilities because it supports more networking protocols, while Unity is a better choice when it comes to Heterogeneity whereas Unity is available for Macs, PCs, Nintendo Wii and iPhone [4].

Zulfiqar and Muhammad proposed a score-based framework to help game analysts and developers identify the best available game engine for serious games and gamified applications. Authors of this paper firstly made an extensive review and evaluation of the existing literature of engine selection. Based on that review they introduced a more general framework to evaluate game engines using score-based framework. This framework identifies three main groups of attributes to use for comparison with respect to their importance for gamification and serious games. The first group includes the most important attributes which are: 2D/3D support, deployment platforms, development platforms and licensing costs/terms. The points for each of these attributes is 25. The second group includes attributes of graphic rendering Audio/visual fidelity, artificial intelligence, physics and networking. Each attribute of the second group is worth 5 points. CAD platform support, availability of import/export of assets, world (level) editor availability, content creation, scripting languages, learning curve and accessibility are included in the third group of attributes where each one of them has a score of 1 point. Using this evaluation framework, the developer can choose among available engines to program a game [24].

Pavkov, et al., proposed a criteria which could be considered before selecting a game engine for serious games. The process of serious games creation is based on the methodology SADDIE which includes Specification of the game, Analysis, Design, Development, Implementation, and Evaluation. Five different engines were evaluated in this research: Adventure Game Studio, Construct 2, e-Adventure, GameMaker: Studio, Phaser Editor. The criterias used for the evaluation process of these game engines were divided into five groups: the first group includes basic features of the engine including price, the second group includes support, flexibility, interoperability, and usability. Another third group includes engine system requirements and installation. Functionality and the ability to export are the fourth group. The last group includes multimedia support and working environment. Authors compared those five game engines to choose the best of them for developing educational serious computer games, based on the approved the GameMaker: Studio was chosen as the best choice for developing whereas It's a very affordable engine with intuitive graphical interface and simple mode which facilitate game development, and it is suitable for beginners in game development [11].

Politowska, et al. explored and investigated game engines and compared them with the traditional software frameworks, they conducted a statistical analysis on a dataset of 282 game engines and 282 frameworks on Github to distinguish the similarities and differences between them. The statistical analysis took into account many software engineering related questions and measurements like the static characteristics (programming language, functions size, and function complexity), historical characteristics like versions control and project life cycle, and community characteristics. Results show that the most used languages in game-engines belong to the C family: C, C, and C#. While for frameworks, JavaScript, PHP, and Python are the most used languages. Engines are usually developed via their main programming language. However, to make it easy for the game developers during the design, production, and testing, the engine's developers add some scripting capabilities to their engine. For example, Unity is mainly written in C but offers some scripting capabilities in C#. However, Frameworks rarely offer scripting capabilities where their products are often developed using the same language. For version controls and history characteristics, Frameworks are released more often than engines and more commits made regularly [15].

3. Research Method

The method used was the Software Architecture Analysis Method, explained below, the project was a small 2D game. Notes were taken during development, questionnaires were filled in and a final interview was held to gather data which was later analyzed.

3.1. Research Questions

The research questions are defined below, together with their explanations and the motivation behind them. As well as giving an understanding on how we planned to answer them. Answering these were the main objectives of this thesis.

RQ1: How is game architecture affected by picking one engine over another?

Game architecture compared to game engine architecture is how elements of the game engine, work together such as sound files, moveable objects and any aspect the developer is working with, whereas game engine architecture is the different aspects of the engine, the physics engine, rendering engine, sound engine among some, work together and are structured [10].

With this question we hope to give a basic understanding on how picking one engine over another can have an impact on the game architecture as well as giving the reader an understanding of game engine terminology and game architecture.

With this, the reader will be able to get a chance to better understand how game engines work, how different engines are different and how to more easily evaluate a game engine based on what it offers and how it meets the requirements for your project. The idea was not to compare all game engines but instead give a more broad understanding of common practices in games, include some terminology to teach the reader about game design and allow them to read up on further studies.

RQ2: How does choosing a game engine affect the structure of the overall game code?

Will it matter what engine you chose when it comes to coding? We expect the code to be different as godot uses GDScript while Unity uses C#, but with this we hope to find out whether or not the choice of game engines impacts the architecture of the game code and how different components communicate with each other via code.

RQ3: To what extent can game engine components in each game engine be reused?

By components, we refer to the various parts of the game engines which are unique to that engine to see if picking one game engine over another has an impact on the reusability of the components. The idea is to measure the usability among different components and whether or not you have to write a lot of code or if the engine allows you to use predefined components to do things for you, and whether one solution to a problem can be used multiple times.

As well as researching on how well the components in the game engine matches the scenarios we have set up for our game, which will limit the overall concepts to 2d-platform games.

3.2. Software Architecture Analysis Method

The method used to plan the iterations and comparing notes was called Software Architecture Analysis Method (SAAM) [5] it is a method used to determine how specific softwares quality attributes were achieved and what impact the future changes will have on the quality attributes based on presumptive cases studies. Some of the quality attributes that can be utilized by this method are maintainability, modifiability, flexibility & reusability [14][5].

"SAAM has six activities: scenario development, SA description, scenario classification and prioritization, individual scenario evaluation, scenario interaction, and overall evaluation" [16] as explained earlier, a scenario is like a scene or a way to describe a certain described point in the future or as Notten [20] said:
"Scenarios are consistent and coherent descriptions of alternative hypothetical futures that reflect different perspectives on past, present, and future developments, which can serve as a basis for action."

Since the goal of the project was known, it was possible to split development into stages of completion, in the case of game development, a game view is called a scene and by splitting the game project into smaller tasks or states it would be in, it is possible to call each stage or iteration, a scenario.

The main focus in the beginning was to develop the scenarios, this was done by splitting the end goal into 6 different iterations or stages as described in the next chapter (3.3). The reason for having 6 iterations was that the time frame allowed for 6-8 weeks of development and having a development session once per week, and upon splitting the game up into separate stages of completion with roughly the same amount of tasks, it turned into 6 iterations. This fit well with the plan of having one development session per week as it would allow us to complete the game in time. The time frame was based on the thesis deadline and allowing enough time after development for analysis and writing.

The end goal would be to perform a scenario evaluation for each iteration and then performing an overall evaluation on all the iterations and components used in the project. With this, it would be possible to achieve an understanding of how the architecture looks like in the different game engines as well as some of their quality attributes which then can be used for studying the reusability of various components.

By using the SAAM it helped compare the engines as the overall project has been made into iterations and smaller tasks (scenarios) which allowed for a more in depth study of the notes taken of each iteration to more precisely compare the elements of the engines.

3.3. The Project

After deciding what method to use for gathering data for the thesis, a project was planned. The goal of the project would be to help answer research question one and two by providing data and notes for two separate game engines which could be compared to analyze the difference in game engine architecture as well as code structure.

Since a comparison needs at least two objects to base the comparison, two engines were picked to compare, Godot and Unity. Then the idea for the project was to develop a simple 2D game using the two engines. Each resulting in the same game, but that it would be developed in a different engine. The game was developed through six iterations to have enough time to study the engine aspect that the iteration revolves around e.g. physics, collision and user input. And to have the time to analyse the answers received from the questionnaires. To help speed development up and have more data to base the thesis on, it was decided that two teams would be created each having two members, this way the game could be developed by two independent teams allowing for a more separated view on the engine and reduce the risk of mixing knowledge. The reason for having two teams was that the comparison would only be between two engines so having more teams could make the comparison uneven. The reason for having two members per team was that another group of students were working on something similar and it was decided that a collaboration would yield the best result as it would give us more data to work with.

One team, consisting of the authors of this thesis, worked on developing the game using the Godot game engine while another team consisting of two other students worked on developing the game in Unity, these students were (Micaela Gustavsson and Rasmus Flomen) who worked on a separate thesis regarding user experience also in the Unity and Godot game engines.

The teams met weekly to work on a iteration and discuss findings and write notes, Each team set up their development environment and then developed the same iteration in parallel on their respective game engine to decrease the risk of forgetting knowledge regarding the game engine, and so it would be easy to compare notes per iteration as each iteration would be the same between the two teams. When working on the scenario, one person was doing the actual development on their computer while the other person was responsible for taking notes. The task of taking notes and doing the development would shift every week so that it was not the same person taking notes each week. This helped ensure each participant got some varied notes to work on and helped us know if some aspects weren’t covered by comparing notes from week to week.

Once each team was done developing an iteration, both teams would meet to gather each other's notes taken during the session. A questionnaire was given out which each team would fill in. This step was expected to give us valid information regarding the game engines architecture.

3.4. Iterations

The iterations used for the project are defined below with sub-tasks to break the work down even further. In other words the iterations are a representation of the steps that we followed to develop the game, and analyse the results that we got using Godot and compare it to the results that the other team got using Unity.

These were written in collaboration with the two students mentioned earlier, who were focusing on developer experience. The reason for these being written with them was to ensure everyone worked on the same things at the same time and that the end-result would be as similar as possible visually and functionality wise. Doing it at the same time and in the same iterations also allowed us to compare notes afterwards about what design methods were used for each iteration, what components were used and how they were used.

Iteration 1 - Environment Setup
This iteration is not as interesting to this research but since the iterations were decided together with the two students working on another thesis, this iteration has been kept.

Task 1	Each team should install and configure their respective game engine on each of their machines.
Task 2	Each team should be able to export or build the game, even if it's empty. (This is to ensure the environment is setup and can run)
Iteration 2 - Game Skeleton Here both teams get to test working with the engine as a beginner, setting up a very basic game with just some platforms. This helps by introducing some of the elements of the game engines.	
Task 1	Add a game area with a blue background.
Task 2	Add platforms to the game following a predetermined layout.
Task 3	Add a checkered flag at the top right of the game area (this is a goal for the person playering to reach).
Task 4	Add a coin counter in the top middle of the screen. (It should just be visual and not update).
Iteration 3 - Player movement: This introduces user input and moving objects	
Task 1	Add a player to the game with an image.
Task 2	Add capability for horizontal movement on user input. (User input is the left and right arrow keys on the keyboard)
Iteration 4 - Physics and collisions: This introduces physics by making the player able to fall and jump as well as a basic form of collision handling.	
Task 1	Apply physics to the player, meaning the player should start falling down when the game is run.
Task 2	Make the player object "jump" when the user presses the spacebar on their keyboard.
Task 3	Collisions with the ground shall be set up so that the player doesn't fall through the floor.
Iteration 5 - Interaction: This introduces a new aspect of collision handling by allowing you to interact with objects in the game	
Task 1	Coins(images) shall be placed around the game map.
Task 2	Make it possible for the player to collide/hit the coin objects.
Task 3	When the player collides with a coin-objects, the coin should disappear.

Task 4	When the player collides with a coin-objects, the coin counter shall update and the counter should go up by one.
Iteration 6 - Sound Effects This introduces sounds allowing us to interact with the sound aspect of game engines.	
Task 1	Introduce sound effects to the game when colliding with coins.
Task 2	Introduce sound effects to the game when finishing the map (Colliding with the finish flag).
Task 3	introduce background music while playing the game, this should just be any music which plays when the game starts and keeps looping.
Task 4	When a player collides with the checkered flag the game shall reset/start over

3.5. Method of gathering data

The following questions were filled in by each team after each iteration, they also have a short explanation as to why they were picked and why they were important for the project.

1. **How long did it take to complete the iteration?**

This question was used to give an understanding of the complexity of the iteration, and the difficulty of getting part of the engine to work together. It is mostly there to give a small help to evaluate the iteration and not a main talking point as it depends a lot on user experience.

2. **Did you face any issues with the game engine?**
For example, you couldn't build, got errors or things broke in unintended ways.

This question was made in case anything went wrong due to design flaws. for example if some things didn't work together as the user expected or if a mechanic had a flaw in it.

3. **Did you face any limitations of the game engine? Ex, the engine didn't support what you wanted to do.**

Knowing whether or not the user experienced conflicts between components or if the components had certain limitations is important to answer the research questions since it indicates that there are flaws or limitations in game engines which the user has to work around which would impact the development of the game.

4. **Did you have to spend a lot of time searching for information to solve the tasks needed to complete the scenario/iteration?**

Even though this question isn't as relevant to game architecture, it is still helpful to know whether or not there is the same amount of knowledge available among different game engines. As that is a good indicator that choosing one engine over another would have an impact on the project, for example, if one engine is highly advanced and can be used for some very sought after games or simulations, but lacks in documentation, picking it over one with better documentation might delay progress.

5. **Do you feel it was straightforward to complete the tasks in the scenario/iteration?**

   This question gives an indication on how different components and parts of the engine work together to give an idea of the usability of the different components which helps stating the importance of picking the right engine for your project.

6. **What parts of the game engine did you utilize to complete the scenario/iteration?**

   The answers to this helps us do our comparison between the two engines. Allowing us to know which parts were used in which iteration narrows it down further and allows us to compare the documentation from both engines and finally discuss with the members of each team how they used different parts to complete the tasks.

7. **What's the different components named which you used and what do they do?**

   This allows us to compare the different components used in each iteration, get a basic understanding of what they do and what they were used for, and allow us to research further into the components on the respective documentation, to summarize the difference between the two engines.

3.6. Final Interview

After completing the game project, we, the authors realized that a few of the questions in the questionnaire were focusing more on developer experience rather than the engine's architecture, they were still useful to the research, but wanting to make sure enough data was available on the more important subject, game design and architecture as well as reusability, 3 additional questions were formed and a final meeting was held. The team working on Unity was interviewed and the authors who were working with Godot, answered them in writing instead.

The answers to the below 3 questions, together with notes taken during development will help answer the third research question by making it possible to detect things which were reusable.

1. **Were you able to reuse components from one iteration in another iteration?**

   Developers in general and game developers specifically tend to reuse the same components multiple times in their project according to their needs. So we wanted to measure to which extent each engine supports the reusability of the same component

2. **How does Unity compare with Godot when it comes to handling nodes? Is it called something else? Does it work differently?**

   This question was added as we felt the system in Godot was clever because it ensured all nodes had a specific usage which could help to decrease confusion for the end user. And since it's such a core concept of structuring objects in the game we wanted to know whether Unity had a similar approach.

3. **How are scripts applied to nodes, and how do they interact with other nodes or scripts?**

   This is more of a specific question, but it's important to know whether it's similar to Godots handling or whether or not it's done differently.

We also took the chance to go over the other team’s answers to minimize any misunderstanding. We also talked about how different things were done, such as importing sound files, images, placing ground objects in the game and handling jumping mechanics.

3.7. Prior knowledge

All of us had studied programming and software architectural courses e.g. “Introduction to Programming in C++”, “Object oriented programming in C++” and “Introduction To Software Design and Architecture” so all of us are aware of programming and software architecture, but none of us studied courses that are dedicated only towards game development, but some of us touched the area of game development while studying the object oriented programming course since the project of this course was to build either a game or a desktop app.

Since everyone had the same base education and we were focusing on how the overall structure of game code, e.g. how different scripts interact we felt setting a code standard wasn’t necessary.

4. Analysis and Results

This section will walk you through each iteration and show findings and observations from the team working in godot, in Chapter 4.2, there will be a summary of commonly used components from each engine as well as observations and explanations based on the data gathered from each team.

4.1. The Project

Since this is a summary of the notes and observations taken by the godot team, it is written in the godot team's perspective, "we" refers to the Godot team.

The reason for only having a detailed walkthrough for the godot engine is that the process in godot was found to be very similar to the process used in unity and therefore it is enough to compare the results. All notes taken during iterations for the godot team and unity team, can be found under appendix 8.1 and 8.2.

4.1.1. Iteration 1

For the first iteration, our goal was to setup the environment to further develop in, since this was related to user experience which we are not interested in researching, we felt this iteration was irrelevant to us, but since the other team required some subjects to monitor while they were installing the godot engine, we still took notes and screenshots to help the other team get their data.
So in this iteration, we've just downloaded and installed the engine, and wrote some notes supported with some screenshots to aid the other team.

4.1.2. Iteration 2

For the second iteration, we started off with discussing the game layout, the actual location and sizes of the different platforms which we were planning to create in the game. After this, we started the program and we began testing some things with the game engine, mostly just getting our bearings and figuring out where different buttons and menus were as well as what they were called. We looked into the right tab, called "Inspector" where there was a button which looked like a file and a plus, upon clicking this we were presented with all the different components Godot had to offer, such as Textures, Animation, Audio, Shapes, Input and lot more. After getting an understanding of how the engine looked and worked, we started drawing up the graphics needed. We did this in Microsoft Paint as we did not feel it would be relevant to use a more advanced tool to do this with. Further work continued with us placing out the platform images in our scene to create the pre-decided layout.

After the layout was done we added a RichTextLabel node which is used to display text, we positioned this in the top of the screen and changed the text to "Coins: 0" to indicate the user having 0 coins.

We also added an image which looked like a black and white checkered flag which would be the objective to reach, the current game and layout looked like the following

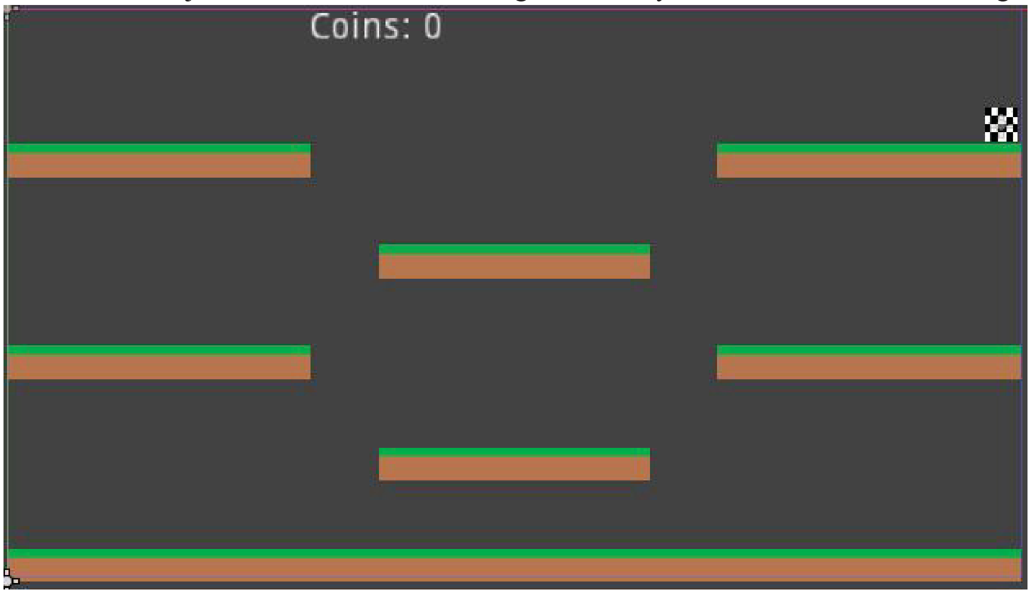


Figure 2 - Game scene layout

4.1.3. Iteration 3

Here we set up the sprite for the player object similar to how we did with the ground objects, we figured out how to add scripts to the player object and began writing the code for making the player move.

In Godot, to take input, you have to define input types in the setting page, by setting a name of the input and a key to activate it, such as "LEFT" and "RIGHT" for the left and right arrow keys. Then via code, you make a function called "func _input(event):" where you check whether or not the input is pressed, like this:

```
func _input(event):
        if Input.is_action_pressed("LEFT"):
                set_axis_velocity(Vector2(-speed,0))
        elif Input.is_action_pressed("RIGHT"):
                set_axis_velocity(Vector2(speed,0))
        else:
                set_axis_velocity(Vector2(0,0))
```

What this does is it applies a horizontal force when left or right input is pressed, to move the player left and right.

4.1.4. Iteration 4

For this iteration, we had to make it possible for the player to jump. We did this by adding an input called "JUMP" which was set to trigger when the spacebar was pressed. Additionally, we had to make sure the player only could jump if the player object was standing on top of a platform.

The way to solve this in Godot, was to add two child nodes to the player of the type Raycast2D. We placed one on the bottom left and right corner of the player sprite, and positioned this to point downwards, like in the image below:

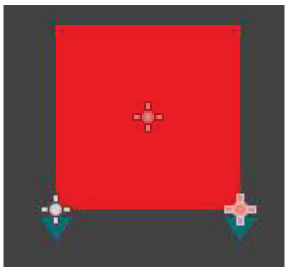


Figure 3 - Player node and raycasts

As you can see, the arrows point downwards but the ray is not that long, which it doesn't have to be, just enough to be able to see whether or not the player is standing on the ground. We experimented a bit with the length to make sure it worked.

What these rays do is you can call a method on them, called is_colliding() to check if they have anything blocking the line.

In our _input method, we added the code below

```
if Input.is_action_pressed("JUMP") and (raycast_left.is_colliding() or
raycast_right.is_colliding()):
        apply_central_impulse(Vector2(0, -jump))
```

What this does, is that it checks if the user presses the spacebar key and that the raycasts are hitting something, indicating that the player object is grounded and applies a vertical force upwards with -jump, jump being a variable defined earlier in the script with a value of 700.

The variables jump, raycast_left and raycast_right are defined like the following:

```
var jump = 700
var raycast_left
var raycast_right
```

and set in the scripts _ready() method. This method is called when the object is created, which is it when the game is run, like the following:

```
func _ready():
    raycast_left = get_node("RayCast2D_LEFT")
    raycast_right = get_node("RayCast2D_RIGHT")
```

We didn't have to create these references and instead we could have just used the get_node method each time, but we read that creating a reference to objects is better for performance.

At this point the player wasn't falling down so we had to change some things. We added a RigidBody2D child node to the player sprite which would make it fall down due to gravity, and a CollisionShape2D which would make the player collide with objects.

Due to how godot handles collision, we had to define some collision layers, one for the player and one for the ground. Then we added a CollisionShape2D to all the ground objects and set their layer to "Ground" and set their mask to "Player" we did the opposite for the player which in turn would mean the Player would only be able to collide with other colliders on the "Ground" layer.

After running the game we realized that nothing visible was happening. This was caused due to the RigidBody2D being a child node and not the parent node, meaning only the child had physics applied to it. We switched places between the Sprite and RigidBody2D node and now everything moved with the RigidBody2D properly.

4.1.5. Iteration 5

The goal of this sprint was to add coins to the game which the user could collide with or "pick up" and add it to their score on the top of the screen. This introduced a sort of collision which allows the user to go through objects but allowing that player or object to know that something hit it.

In godot, this was done by drawing a coin in an external program and importing that into godot. After this we created an Area2D and added a child node of the type Sprite and a CollisionShape2D which we set to a rectangle. After this we copied the coin object into different places on the map.

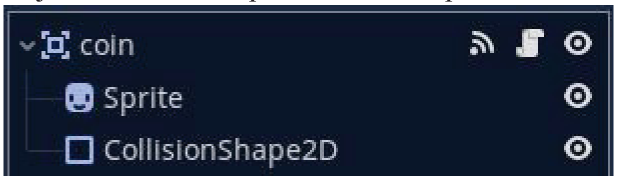


Figure 4 - Coin and child nodes

After some searching on the documentation, we found out that to tell the player object that it hit a coin, we had to add a script to the coin which has a function for handling objects entering the collision area:

```
func _on_coin_body_entered(body):
        if body.get_name() == "Player":
                body.add_coin()
                queue_free()
```

This calls the add_coin method on the player object and a method called queue_free() which removes the coin, then in the player script we wrote the method called add_coin() and set up some variables to be used.

```
var coins = 0

func add_coin():
        coins += 1
        coinSound.stream
        coinSound.play(0)
        get_parent().get_node("RichTextLabel").text = "Coins: " + str(coins)
```

What this does is increase the coins variable by 1, tell the coin sound to play once and update the RichTextLabel with a next text showing the new value of the coins variable.

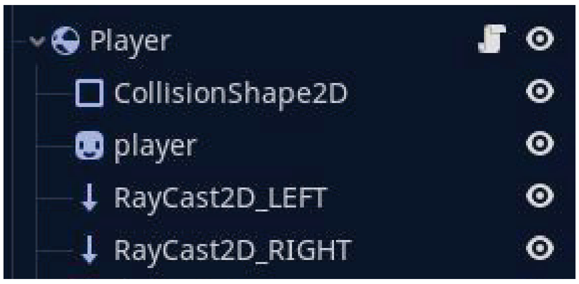


Figure 5 - Player node and child nodes

4.1.6. Iteration 6

The goals for this iteration were to introduce sound effects when the player collides with a coin, the flag and add a music which plays on repeat when the game starts, and finally have the game start over when you collide with the checkered flag.

We did this by creating a sound effect in an external program as Godot didn't support making sound effects in the editor. We made a sound for when the player hits the coins and another one for when the player hits the checkered flag. We also downloaded a piece of music from the internet. The sounds used weren't important to the project.

We added three AudioStreamPlayer2D In the player object, one for each sound. The AudioStreamPlayer2D is a node that allows you to set a sound to play when the play() method is invoked via code. At this point, the player node had the following child nodes:

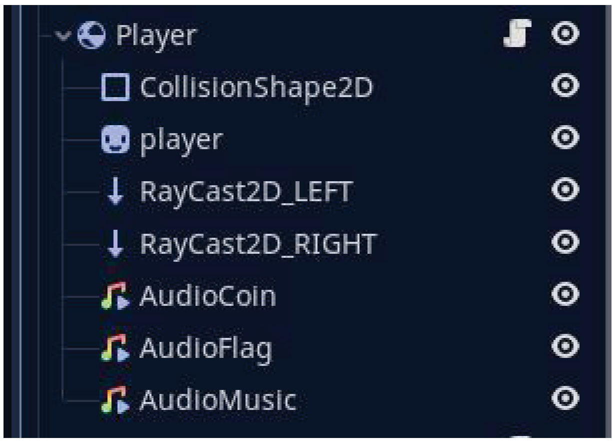

```
Player
    CollisionShape2D
    player
    RayCast2D_LEFT
    RayCast2D_RIGHT
    AudioCoin
    AudioFlag
    AudioMusic
```

Figure 6 - Player node and more child nodes

Since the music was supposed to play on startup and loop, we changed the node to do this by toggling an option called autoplay in the inspector tab:

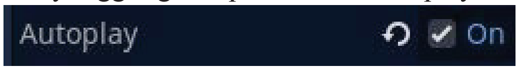
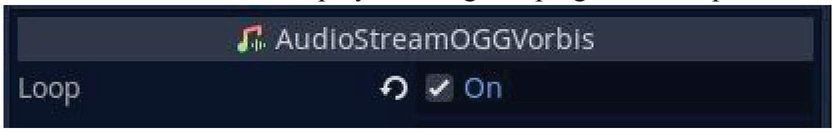

Autoplay ✓ On

Figure 7 - Autoplay

and set the audio file to loop by enabling "looping" in the imported audio file.


AudioStreamOGGVorbis

Loop ✓ On

Figure 8 - Loop option

After this we defined the variables for the audio in code like below

```
var coinSound
var flagSound

func _ready():
        coinSound = get_node("AudioCoin")
        flagSound = get_node("AudioFlag")
```

Then in the add_coin method we added the following two lines which would cause the audio to play when the player hit a coin

```
func add_coin():
	coinSound.stream
	coinSound.play(0)
```

We setup a flag object similar to how the coin was setup and made a "on?body_entered" method on the flag script:

```
func _on_FinishFlag_body_entered(body):
	if body.get_name() == "Player":
		body.flag()
```

This would call the flag() method on the player which simply played the sound and reset the game with one line of code:

```
func flag():
	flagSound.stream
	flagSound.play(0)
	get_tree().reload_current_scene()
```

4.2. Components used in the project

This chapter will summarize the main components used in each game engine to complete the project, explain what they do or can do, and explain how they were used by the two teams. As well as having a conclusion in the end comparing differences and similarities between the components. Due to the large amount of components used, additional component descriptions can be found in the appendix 8.3

4.2.1. Components used in Godot [7]

Scene

A representation of Nodes to portray the game, the game needs at least one scene but can have more. We had a single scene where we put all nodes relating to the game in. A scene can also be a set of nodes which can be placed into other scenes, this allows you to create one "scene" which acts as a node and can be reused in other scenes. We did this for the coins.

Node

A building block with attributes which can contain more child-nodes (Nodes attached to the main node). These nodes have unique attributes depending on the node type, examples coming further down. Nodes can be moved around in the scene depending on where you want them to be. This can be done in the editor or via code.

Scripts

A file or class where you write code which can be attached to nodes to interact with the node and child nodes. It can access methods on the node as well as signals.

The scripts in Godo, were written in a scripting language called gdScript which was made by Godot, this language is similar to Python.

A script was attached to the player node which listened to user input and in turn moved the player from left to right as well as jumping when you pressed the spacebar on your keyboard. The player script also had a ready method to set up references to some raycasts, sound nodes and the rich text label node. Additionally two separate methods were written, called add_coin() and flag().

To allow the coins and flags to be interactable, each coin and flag had a script on it which in turn would call on a method stored on the player to add coins and restart the game.

Sprite

A node which has an image which can be shown in a scene or attached to another node. These types of nodes were used for the player object, the coins, the ground platforms and the end flag.

Raycast2D

"A RayCast represents a line from its origin to its destination position" [7] They are used to check whether anything is in between two points. In the case of the game, it was necessary to check whether or not the player node was positioned on top of the ground. In the case of being grounded, the person playing was allowed to jump with the space button. To achieve this, a Raycast2D was placed on the left and right side of the player pointing downwards, and then in the player script, it was possible to check whether or not the rays were intersecting with anything.

4.2.2. Components used in Unity [8]

GameObject

A gameobject is similar to a node in godot, a gameobject can have components attached to it which adds functionality to the game object similar to how in godot you would add nodes as child nodes to add functionality such as collision.

Scripts

Scripts in unity are similar to scripts in Godot, they are used on game objects to add functionality to them and can access methods on the components attached to the game object.

Note from other team: *"A small piece of code that completes a specific task. in this case movement. We only used it to move the player object left and right."*

Sprite

A sprite is an image often used in 2D games, it can be stretched or rotated and was used for the different objects in the game, such as the player, coins, finish flag and floor objects.

Rigidbody2D

This component can be added to 2d Objects or sprites to make it affected by the physics engine, allowing it to be affected by gravity. It can be manipulated by scripts to be moved around. This can be done by applying a force to the RigidBody2D in a certain direction. This was used to apply physics to the player object (make the player fall down) and was also used to make the player jump.

Raycasthit2D

This component works the same way as it does in Godot, and was used to check whether or not the player object was standing on top of the ground to be able to trigger a jump.

4.2.3. Comparison between components

Godot and Unity are both advanced game engines with a lot of features to allow the developer to more quickly develop their game. They both offered similar features to complete the tasks but still had some different things between them.

The main one being how different objects are handled. In godot each node has a specific usage and cannot be edited similar to how unity allows you to, but this didn't stop us from completing the tasks as they had a lot of different nodes for all potential purposes. But since each node does something else, you have to either constantly reference child nodes or create references to them and if you don't place the right node as the parent, it might cause conflicts as it did for us when the RigidBody2D was set as a child node instead of parent node.

Additionally, each node has a lot of attributes which are not edited and simply stay the same between all nodes:

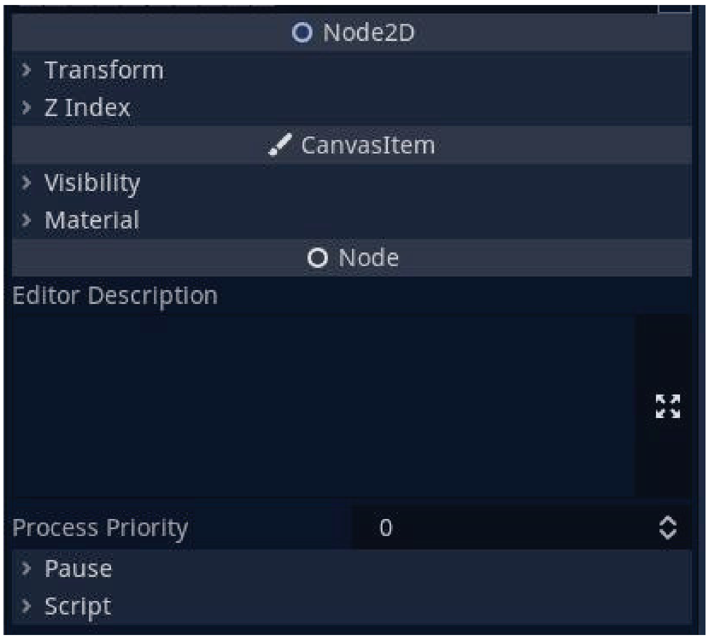


Figure 9 - Common node attributes

This means a lot of unused information is stored and it clogs up the UI, you also have to select the specific node to see information regarding it which slows down development a bit.

In Unity, you don't have the concept of nodes, instead there are GameObjects [8] which work like godots parent nodes, and instead of adding child nodes you instead add components to that object. This means you can have common information between components and all components can be attached to a single gameobject.

You can also have empty game objects which only contain a transform, which is a coordinate and rotation. Even though this is possible, you are able to reproduce the node system godot offers by creating a child gameobject and attaching one component to each. This means the two ways of handling gameobjects and nodes can be similar but in our implementation it was not.

Overall, it was simple to find the nodes needed, and most objects in the game used the same node types or components such as CollisionShape and Sprite.

Unity offered an even better solution to re-using sprites with it's tilemap, allowing you to use one gameobject to place out your sprites and only requiring you to add one collider to the tilemap for it to apply collision to the whole ground, whereas in godot you had to create a node per ground object.

One thing godot did which was quite interesting, was that you could create a new scene and add a sprite to it as well as a collision shape and after saving that scene, you could use that scene as a node, this was done for the coins and it allowed the developer to edit each node by clicking the parent node, which no longer showed the child nodes, you would have to open the scene to edit the coin, but in doing so you also edited all other instances of that node. This helped improve reusability by a lot since the developer could do the change in one place and it would take effect everywhere.

Unity had a similar approach where you could create a gameobject and save that as a prefab. This prefab could then be copied around in the scene to create linked copies of that gameobject, and editing one would affect all of them.

5. Discussion

This chapter will go through the findings and differences of the game engines that we, the authors of the thesis found as well as discussing the answers to the research questions by looking at the analysis and results from the previous chapter.

There will also be a section discussing validity threats and how they might have affected the result.

5.1. RQ1. How is game architecture affected by picking one engine over another?

Overall both Unity and Godot have a lot of common core concepts and share similarities in game architectural structure, such as scenes, objects or nodes, scripts and how the different components communicate with each other. But while Unity allows you to have a lot of components on single gameobjects, Godot relies on a unique node based hierarchy to achieve similar functionality. Additionally, collision detection goes in the opposite direction forcing you to write more code which in turn increases the complexity.

By comparing Unity's gameobject system with Godot's Node system it's possible to come to the conclusion that they both can offer the same implementation but that the approach to do so is different. In Unity you can create an empty gameobject which only holds the transform data which is the rotation and position in 2D or 3D space. This was done by the team working on the Unity version of the project. With this object, they attached components to it such as a Sprite to get an image on it, Rigidbody2D to get 2D physics applied to it, a boxCollider2D to add collision and a script to handle movement, collision handling, user input and restart of the game. This way, you only have one object to keep track of and all components are visible under that object.

In Godot on the other hand, you had to create a node of the type RigidBody2D which was renamed into "player" to make it easier to find in the list of nodes. After this, similar to unity child nodes with similar functionality as the components in Unity were added to the player node. And in the end the representation of components was the same as the one in Unity.

One of the main differences we noticed when comparing notes was that Unity's collision handling was in a way, simpler. Every collider component collided with each other and you did not have to set up any rules for it. whereas in godot, you had to set each object to a collision layer and define what layers it would collide with. layers which were not set to collide allowed the player to go through it and in turn it could detect something entering the collider area, and the role of detecting collisions was given to the coins in the case of the game we developed. Compared to unity where objects which the player should be allowed to pass through and be able to detect you could set the collider to "is Trigger" which meant it didn't collide with objects, and by doing this on the coins you could detect the colliders overlapping in the player code instead of in the coin code.

So in summary, game architecture and design patterns are similar but vary slightly between different engines. The end result can be similar to that of another engine and so can the design be. The thing that varies most in the approach to get there.

5.2. RQ2: How does choosing a game engine affect the structure of the overall game code?

Comparing the way unity and godot allows you to add scripts to objects we see that they both have a similar approach but it is not fully the same, meaning there will be an impact on the game code architecture. This impact will mean you have to think differently about how scripts interact with each other.

Both Godot and Unity call their code components scripts, in Unity, these scripts can be attached to any object and be generic scripts which do something like moving an object, you can also have multiple scripts on objects whereas in godot, scripts are made for specific nodes and cannot be reused between different nodes, there can also only be one script per node.

A thing mentioned in 4.1.5 and 4.1.6 is that godot adds another level of complexity by reversing the role of detecting collisions. In unity, you would put the code to check for entering a collider in the player object and in the case of coins, remove the object you collided with and step the coin counter. Whereas in godot it's the coins responsibility of detecting the collision and then telling the player that the player collided with the coin. And it's also the coins responsibility of removing itself

By only allowing one script per node and not allowing you to reuse scripts on other types of nodes, you limit the flexibility of the scripting system. And by having the role reversed on certain things you increase the complexity by having to let the user keep track of more scripts.

5.3. RQ3: To what extent can game engine components in each game engine be reused?

Unity and godot both allow you to reuse engine components which increases development speed by not having to redo things multiple times. By comparing game engine features and components we can get an understanding of how components can be reused and how it affects the progression of the project. In this project specifically we didn't find any unique features that made one engine better than the other in terms of reusability and instead we saw that both engines offered the same two concepts to help speed up the game design and development.

An example of this is explained in 4.2.2 with Unity offering a prefab system, if you have a game object which you know you will be reusing, if you were to copy this object several times and had to edit it at a later date, you would have to make that change on all game object which would take up much of your time, especially if you use that gameobject in a lot of places. With the prefab system, you can save one gameobject to your assets folder (a tab in the editor showing you the project files) which will save that object as a prefab. This object can then be placed around in your scene or created via code. And any change made to one can be saved and pushed to all copies of that game object allowing you to only need to edit it in one place. Godot has a similar implementation, where you can create a new scene and add your nodes to that scene. And by saving that it turns the scene into a new node which you can then place out in other scenes. Editing that scene will update all the nodes. This also allows you to edit all the child node attributes in one place instead of having to click on each child node.

When building the game map in godot, the team working with godot created a sprite node with the image of the ground and a collider. They then copied that object around to create the pre-designed map. This process was slow but it still didn't take too long due to the size of the scene not being that large. The unity team realized that

Unity offered a tilemap system to speed this up by a lot. How this works is that you create a tilemap object, you define a palette of sprites and then you can “draw” the sprites out in a grid pattern allowing them to quickly build up the game scene. After drawing out the tiles, they could add a tilemap collider which would enable collision on all of the tiles in one go. This process took a bit longer to set up but allowed them to more quickly build the layout of tiles. If the project was larger and required a larger game area or multiple different game levels, this approach could have saved a lot of time. Additionally, using this approach allows you to edit all the images in one place by editing the sprite image. A similar approach to solve that would have been to use prefabs. Looking over the godot documentation [7] we see that godot offers the same solution with their own tilemap system which the Godot team didn’t use.

All in all, each engine offers similar solutions to allow reusability of gameobjects, saving time on not having to redo changes a lot of times and offer a good way to draw game maps which in turn allows you to use one instance of a sprite or gameobject multiple times.

5.4. Validity Threats

The goal of this chapter is to mention things that could threaten the validity of our thesis, such as things that might have been done wrong, limitations or too little data to base our understanding on.

5.4.1. The Game Project

Due to wanting to cover more than one aspect of game engines, the size of the project was held down and restricted to a 2D platform game. Due to this limitation it is possible that some important aspects of game engines and game architecture weren't used or researched which could impact the result. 2D is a popular genre of game and it was possible to cover a decent amount of aspects, as mentioned in 3.3. but the threat is still there.

In the first meeting with the teams a discussion was held regarding what type of game could be made which would both be simple enough that it would not take too long, to not reduce the amount of time available for the study, as well as something which would cover more parts of the game engine.

5.4.2. Oversights in Interview questions

As mentioned in chapter 3.5, after completing the iterations and finishing compiling notes from the teams, it was deemed that more questions were needed to properly answer RQ3. This oversight might have threatened the validity of the thesis by not having the answers during development and instead collecting it all afterwards, this runs the risk of developers forgetting things which could have helped strengthen the answers in RQ3.

5.4.3. Team size and game engines

As stated previously, two teams of two members each, working on two separate game engines. Since there were only two people per engine, the amount of knowledge gathered from the testing sessions could be limited, and since the team members worked together on the same engine and not independently, this knowledge could be further limited.
An improvement to this would have been to have more people. Each working on their own instance of the game, solving issues alone and discussing the results after each

iteration. This way hopefully no one will get stuck on something for too long and it results in more observations of the game engine.

5.4.4. Meeting issues

Due to the widespread virus Covid-19. The meeting rooms closed at Blekinge Institute of Technology, the place of study and where meetings were held. Due to no other place being available the teams had to resort to online meetings to hold the development sessions.

This was not a big hindrance as there was free software available to do this with such as Discord. Though due to bad internet connections these meetings were sometimes slowed down or impacted since it was impossible to share screens or talk with each other.

At one point, someone from the godot team was unable to share their screen, and due to people having work and other plans outside of school, as well as us wanting to follow a schedule of 1 iteration per week, the person taking notes had to instead to the development instead as they were able to share their screen. We don't believe this has impacted the project in a bad way, instead it simply caused a slight delay, but it is still something to include as it is a slight threat to the validity of the project.

5.4.5. Using Ourselves in the experiment

Since we, the authors of this thesis were also participants this might have changed the outcome of the thesis. Had we taken two other participants perhaps the result could have been slightly different, since we were comparing the two engines and collecting answers to the questions from the other team, it slowly helped us get a better understanding of the engine, from the other team. Even though the Unity team were working on another engine, if game engines are similar this helped us slightly.

6. Conclusion

A project was planned to create a game in two separate game engines with two separate teams. This project lasted 6 weeks, each week consisting of meetings to develop the game in set task intervals and discuss findings. And in the end each team had managed to create a simple 2D game, the project was overall successful as it clearly showed the importance of doing proper research before picking a game engine and clearly showed that even though game engines are similar, they still wary enough to make it more or less difficult depending on what project or game someone is developing.

The project also helped answer our research questions, by showing that Unity and Godot have a lot of common core concepts and share similarities in architectural structures and design patterns but also had different ways to let the developer solve tasks, such as Unity using game objects and godot using nodes and handling collisions in another way. Each engine also had some ways to speed up development by offering the developer to reuse components by creating prefabs in Unity and scenes in Godot. And lastly by comparing the architecture of the code it was possible to see that each engine had a different approach to allow code or "scripts" to interact with each other, Unity being more keen on having everything in one place and Godot needing scripts to be spread out more between nodes.

6.1. Future Work

A future study which could be interesting to the authors or reader would be to delve further into other aspects of the game engines as there are so much more to them which weren't studied in this thesis. Having studied the documentation of both Godot and Unity, there are a lot of interesting and unique aspects to them which makes the work for a game developer much simpler.

These include but are not limited to, particle systems; a way to display small objects using the GPU such as sparks, leaves, smoke and explosions.

Networking; Ways to let people play the same game on different devices, making turn based games such as card games, or even large scale projects with thousands of people playing together.

Compute Shaders; A program which runs on the computer's graphics card and allows you to render things in parallel and give you a large performance advantage. AI & Pathfinding; Teaching enemies in games to play by themselves and navigate around objects.

Additionally, it would have been interesting to study more game engines than just the two used in this project, that way it would be possible to answer the research questions more precisely by having more data to compare and found out whether or not all game engines work similar or if more things are shared between them.
It would be beneficial to include more participants in the project and have each participant work on the game by themselves. This way you could know whether or not there were multiple solutions to problems.

Another interesting thing to study in the future would be the performance of different engines in different scenarios.

7. References

[1] Preshing, J (2017, December 18). "How to Write Your Own C++ Game Engine" https://preshing.com/20171218/how-to-write-your-own-cpp-game-engine/ [Accessed July 20th 2020]

[2] Kissner, M (2015, October 27). "Writing a Game Engine from Scratch - Part 1: Messaging." https://www.gamasutra.com/blogs/MichaelKissner/20151027/257369/Writing_a_Game_Engine_from_Scratch__Part_1_Messaging.php [Accessed July 20th 2020]

[3] Lewis, M. and Jacobson, J., (2002). "Game engines. Communications of the ACM", 45(1), p.27.

[4] Petridis, P., Dunwell, I., de Freitas, S. & Panzoli, D. (2010), "An Engine Selection Methodology for High Fidelity Serious Games", IEEE, , pp. 27.

[5] Kazman, R., Bass, L., Webb, M. & Abowd, G. (1994), "SAAM: a method for analyzing the properties of software architectures", IEEE Computer Society Press, , pp. 81.

[6] Rather, M.A. and Bhatnagar, M.V. (n.d.) (October 2015). "A COMPARATIVE STUDY OF SOFTWARE DEVELOPMENT LIFE CYCLE MODELS". International Journal of Application or Innovation in Engineering & Management, Volume 4, Issue 10.

[7] Juan Linietsky, Ariel Manzur (2020) "Godot Docs – 3.2 branch" https://docs.godotengine.org/en/stable/index.html Accessed 9 May. 2020

[8] Unity Technologies (2020) "Unity User Manual (2019.3)" https://docs.unity3d.com/Manual/index.html Accessed 9 May. 2020

[9] Haas, J. (2014), "A History of the Unity Game Engine", Digital WPI Interactive Qualifying Projects (All Years) Interactive Qualifying Projects.

[10] Anderson, E., Engel, S., Comninos, P. & McLoughlin, L. (2008), "The case for research in game engine architecture", ACM, , pp. 228.

[11] Pavkov, S., Frankovic, I. & Hoic-Bozic, N. (2017), "Comparison of game engines for serious games", Croatian Society MIPRO, , pp. 728.

[12] Blow, J. (2004), "Game Development: Harder Than You Think", Queue, vol. 1, no. 10, pp. 28-37.

[13] Juan Linietsky (Jan 14, 2014). "Godot Game Engine FIRST PUBLIC RELEASE!" https://godotengine.org/article/first-public-release [Accessed July 20th 2020]

[14] Babar, M.A. & Gorton, I. (2004), "Comparison of scenario-based software architecture evaluation methods", IEEE, , pp. 600.

[15] Politowski, C., Petrillo, F., Montandon, J.E., Valente, M.T. & Guéhéneuc, Y. (2020), "Are Game Engines Software Frameworks? A Three-perspective Study".

[16] Marín-Lora, C., Chover, M. and Sotoca, J.M. (2020). "A Game Logic Specification Proposal for 2D Video Games". Trends and Innovations in Information Systems and Technologies, pp.494–504

[17] Charrieras, D. & Ivanova, N. (2016), "Emergence in video game production: Video game engines as technical individuals", Social Science Information, vol. 55, no. 3, pp. 337-356.

[18] Akbar Baig, J.J. & Al Fadel, M.A. (2017), "Measuring reusability during requirement engineering of an ERP implementation", IEEE, , pp. 258.

[19] Ringland, G. (1998), "Scenario Planning", John Wiley & Sons, Chichester.

[20] Notten, P. (2006). "Chapter 4 Scenario development: a typology of approaches". Think Scenarios, Rethink Education.

[21] Juan Linietsky (Apr 01, 2016) Godot aims for mainstream https://godotengine.org/article/godot-aims-mainstream [Accessed July 20th 2020]

[22] Ritchie, A.C., Lindstrom, P. and Duggan, B. (2017). "Using the Source Engine for Serious Games."

[23] Partha Sarathi Paul, Surajit Goon and Abhishek Bhattacharya (2012). "HISTORY AND COMPARATIVE STUDY OF MODERN GAME ENGINES."

[24] Ali, Z. & Usman, M. (2016), "A framework for game engine selection for gamification and serious games", IEEE, , pp. 1199.

8. Appendix

8.1. Notes taken during development from the Godot team

8.1.1. Iteration 1

I started by opening my browser and searching for "Godot game engine" the first result came up with a link to godotengine.org and some information about the engine After clicking the link and scrolling down a bit, there's a button called download. Upon clicking that I'm taken to the engine's download page where I'm presented with the options Standard version in 64 bit or 32 bit and Mono Version (C# Support)

Since I wasn't sure what language we would be coding in, and we hadn't decided anything prior to this, I downloaded the standard version, picking 64bit as that matches that of my computer.

Downloading the file was quick and I got a 27MB zip file which I extracted to my desktop. The only thing in the zip file was an executable file called "Godot_v3.2-stable_win64.exe" I thought this was an installer but when opening it I was presented with a menu called "Project Manager" where I can view templates, create or import projects.

I clicked "New Project" and a pop-up appeared asking for a project name and an empty folder.

I filled this in with "Test project" and made a folder called the same and hit create.

To verify the engine was set up I clicked around on the top bar till I got to Project>Export I wanted to try and build the game to make sure I had everything I would need to start. There was an add button so I clicked it and picked “Windows Universal” as that was the OS on my machine. It told me I had to download the export template for Windows so I clicked download and it began. After a while it was done downloading the 433.8 MB template.

After filling in some of the information by just filling in "Test Game" and Company name like it already suggests in the screenshot for the export window, I got confused as to what the GUID meant so I googled "how to export godot to windows 10" Going to the first link https://docs.godotengine.org/en/3.1/getting_started/workflow/export/exporting_projects.html didn't help, it mostly had information for android or linux the other links on the search result didn't help either so instead I tried going back to the export page and adding another export for "Windows Desktop" instead. this one didn't need any input and the export button wasn't grayed out, when I clicked export it asked me where to save and I saved it to a folder. It created a file called "Test Project.exe" which when run does nothing. I expect this is due to the game being empty. Now the engine is set up and I'm able to export, therefore I feel the iteration is done.
Total time: 35 minutes, Finding the download link took 1 minute
Download and unzip time took roughly 30 seconds, Downloading export templates took 10 minutes due to slow internet, Figuring out how to export took 5 minutes, Rest of the time was spent documenting and taking notes.

8.1.2. Iteration 2

We started off iteration 2 by discussing the game layout. After coming up with some basic structure on how it should look, we started drawing the images for it.

We needed 3 images, one blue background, one finish flag and a platform. We took a standard resolution of 1920x1080 and divided by 30 because we wanted the world to be 30 "tiles" or grids wide.

This gave us 1920 / 30 = 64 px, taking the height 1080 and dividing by 64 gave us ~17. The scene would be 30x17 tiles.

We drew some simple pictures to resemble the game scene, one image which was supposed to look like a grass platform and one which looked like a finish flag of some kind.

After completing this we opened up godot and clicked import, this showed us a folder where we assumed we had to place our images so we moved them to there and they were instantly visible in the editor. After this we clicked create a scene on the side bar as it said we had to and we were able to start.

After dragging out the floor image to the game view we noticed a button on the top bar there was a "snap to grid" button, we clicked this and then clicked the setting button and set the grid size to 64px instead of 8. this instantly allowed our tiles to snap to a 64x64 grid, we saw that the tiles were offset weirdly and saw in the inspector there was a offset tab, we clicked this and tested changing the offset to half of the images size, 32 and it snapped properly to the grid.

After this we started placing out the tiles on the grid to match what we designed earlier. After this was done and we hit the play button we noticed the game was zoomed in too far or the screen size didn't fit. After googling for this we saw we could change the resolution in project settings. We changed it to 1920x1080 and it improved slightly but was still off. We noticed under display settings there was a stretch option. we tried enabling this, setting the mode to keep aspect ratio and tried running the game again. now the platforms were all inside the window.

After this we clicked add and searched for text, got a text label which we added, aligned to the center of the screen and changed the text to "coins: 999" we were unsure on how to make the text larger so we googled till we came to this page: https://godotengine.org/qa/3335/label-text-size here someone said to set the scale to increase the size which we did. After setting the scale to 5 it looked much better

8.1.3. Iteration 3

I created very simple png pic in "Paint" this will be our game object open paint resize the picture to 64*64 and fill it with red color, then save it as a png In Godot I included the game object to the project and I pressed on it, and the I chose "Configure Snap" from the shortcut bar that I have above my working area and I fixed the "Offset" for the game object

Now we have game object on the screen and we need to move it So I searched in google for "Moving object in Godot" I found the solution here: https://godotengine.org/qa/25249/how-do-i-make-something-move-solved

Now I need to write the movement code for my object I pressed on the game object and then I choose "Script" from the right sidebar in Godot, then I pressed "New Script", so I got a screen with editor, so now I can write code and I managed to make the object move constantly to the right direction, but I want the object to move right and left based on arrow keys pressing and not by it self So I searched in google for "Get user input godot arrow keys"

and I found how to move the object in this page https://docs.godotengine.org/en/3.1/tutorials/inputs/inputevent.html and I found the following code in the page so I copy it and paste it in Godot scripting editor with some modification

```
func _unhandled_input(event):
   if event is InputEventKey:
```

```
if event.pressed and event.scancode == KEY_ESCAPE:
    get_tree().quit()
```

and I managed to move the object using the arrow keys now

8.1.4. Iteration 4

searching in google for “add physics to player godot” Pressing on the first link, which is a documentation for Godot https://docs.godotengine.org/en/3.1/tutorials/physics/physics_introduction.html We added a staticBody2D to our floor object, it complained and said it didn't have a shape and to add a CollisionShape2D to it which we did. After this it complained again in the form of a warning symbol next to the CollisionShape2D saying we needed to add a shape. We clicked the CollisionShape2D and selected a shape in the inspector drop-down menu. we picked rectangular and changed the size to fit the floor. We did this for the remaining floors and then created some layers in the settings. We created the floor and player and set the collision layer on the StaticBody2D to the floor. On the player we set the collision layer to the player. Then did the opposite for the collision mask, setting player collision mask to floor

we made a function for our player object and we specified if the "Space" button is pressed and one of our RayCast touches the ground or the plattform, the the object will move upwards and we apply force to lift up the object

8.1.5. Iteration 5

We started by creating a new image for the coin. We did this in paint by creating an empty image which was 64x64 pixels and drew a coin. We imported it into godot and moved it into the scene. After this we added a collision to it, we read on https://docs.godotengine.org/en/3.1/tutorials/physics/physics_introduction.html that Area2D "provide detection and influence. They can detect when objects overlap and can emit signals when bodies enter or exit." Which is what we want.

After this we googled on how to check for collision with area2d and found this https://godotengine.org/qa/16577/how-to-detect-a-collision-with-an-area2d this didn't work and with some more googling we found this: https://godotengine.org/qa/9834/body_enter-area_enter-signal-area2d-something-become-visible after a lot of testing we found the issue, when you create the collision method, you have to select the coin, otherwise it won't work This wasn't intuitive and we wasted a lot of time on it. After this we made the coin call the player object, a new method called add_coin in which we increased a counter and updated the text label to "Coins: " + str(coins) we also made it update the counter on startup.

8.1.6. Iteration 6

we started with create new node, save branch as scene, and I added a spirit, then I added the finish flag to the spirit then I added rectangular shape 2d to the flag then I pressed on the node tab on the write side bar and I add bodyEntered to the finish flag then I connect it to finish flag I did the same thing with the coins. So now we have fixed collisions in coins and finish flags. I downloaded the Bfxr sound library because it contains a lot of sound effects for games. I googled how to play sound in godot, I pressed on player object,created new node,I chose audio then I pressed on inspector on the right sidebar and I added the sound effects by dragging and dropping them from the FileSystem to the inspector on the right sidebar to the I did this for both finish flag and the coins. A problem that we faced was that we have or audio files as mp3, and it doesn't work at the beginning, then we tried wav extension and it didn't work either So I converted it to ogg and it worked perfectly fine

8.2. Notes taken during development from the Unity team

8.2.1. Iteration 1

sökte på google: github download, valde första länken choose your unity + download individual -> student. create login. email verification. github login. github verification. download unity hub -> install. manage licenses. activate key from email. go back -> installs -> add -> choose version and modules. download unity -> install -> download/install visual studio enterprise 2019 (auto). new project -> 2D -> project name. Fick leta mellan olika tabs, visste inte vad jag letade efter. file -> Build and run. Execute file in chosen folder. Tyckte det va ganska lätt förutom att hitta hur jag skulle exportera spelet, dels för jag visste inte vad det innebar.

8.2.2. Iteration 2

vi började med att rita våra resurser i paint.

-mark = brunt block med grön top (size 64*64)

-spelare = rött block (size 64*64)

-checkered flag = checker'd flag (size 64*64)

try to find information by going thru the ui and google insted we used the 64p*64p grund tiles to size the screen to a 1920*1080 to find the block place on the x-axis we used black magic math (this method was probably not the fastest one and it took a long time) we have massive problems with getting the right sizing then we found that unity had something called tilemap editor. We had big issues with understanding how to use the tilemap editor. The two key factors for this were poor guides and that unity editor has a cluttered UI after a while we found a youtube video that explains how to use the tile map and its editor that is a part of the unity editor. to use it we first needed to create a tileset witch we did in paint this tile set contained ground tile, player tile, and checkered flag tile then we imported into unity editor where we could specify there size to 64p*64p then we opened the tile palette where we created our one tile palate. Then we used the palet to paint the map. doto the fact that we have the correct size on our tiles the map ended up in the correct size. After this we adjusted the camera to have the aspect ratio of 16:9 and sized it to fit our map. And it worked!!

After this we wanted to add the coin counter (non functional)

step 1: add text this was easy enough even though the menu can be confusing to a beginner.

step 2: place in the correct place. This where wirde sence you need to zoom out really far to see the hole ui layer because it adds on top of everything else on the work space and is much bigger than the map layer.But it corresponds to the map layer.but other then that it was easy with the general side menu where attributes can be adjusted ex: color and font size.

8.2.3. Iteration 3

Task 1 Time: 1:30 min

Open the Tile palette on the side panel and add the picture with dragging from the bottom panel.

Add the picture into the view as before.

This had to be redone, as the script didn't work on palette objects, only on seperate dragged objects.

Task 2 Time: 33 min
Click on the object, didn't find anything visually useful.
Tried right-clicking didn't do anything at all.
Looking at the menu, for something obvious, didn't find anything.
Searching on google for how to add player movement with arrow keys.
Searching how to add script to object
Asset → create → c# script
Opened Visual Studio
Dragged the script to the object on the grid
Updating the script with input found on google
Was wrong as it only worked on the grid
Redid task 1, by adding the object without the palette.
The script was now added to the object.

8.2.4. Iteration 4

task 1
started by searching on google on how to make objects jump.
the we fund a link with a lot of code so we copy pasted it
then we modified the code.
then we tried to run the game
there was an issue with rigidbody2D
google was needed and youtube
rigidbody2D is to allow the object to jump
We also need to make sure that we only can jump if the player is grounded.
it works but now our object is falling forever so we need to add collision to our tile map so we found a youtube video for this. This video shows an easy and simple way to do this and now we found that the jump didn't work since we didn't use velocity instead of ?? and now we found that we only could jump once since our grounded funktion didn't work properly.

8.2.5. Iteration 5

Öppnar paint för att skapa coin objektet Skriver i pixlar/storlek på de precis som förut 64x64
Väljer färgen gul och fyller i hela och sparar. Går in i unity → import new asset → väljer myntet
Söker på unity add collision action
Får upp en dokumentation från Unity
Lägger in ny komponent till myntet som har lagts ut -> box collider 2d
Lägger ut fler mynt och gör samma
Kollar upp box collider 2d i dokumentationen som hittades
Söker på unity onCollisionEnter2D
Får upp Unity dokumentation
Lägger till funktion till spelaren onCollisionEnter2D
Spelaren försvann?
Funktionen kommenteras ut
Kollar på de andra exempel, men går snabbt över till en youtube video som är unity official tutorial (detecting collisions)
Lägger till nytt script till spelaren = Destroy Coin
Väljer sen att strunta i de och kommentar därför ut koden som kommenterades innan

Då han trodde att den som fanns inte hette MonoBehaviour utan att de behövdes skapas men som de inte behövdes, då den hette de

Testar spelet, spelet loggar ut de som fanns i testet

Lägger in en if-sats som kollar om objektet kolliderar med coin

Testar spelet

Lägger in i debuggern vilket objekt den krockar med

Men fel object skrevs ut, korrigering från mig om att ändra till col.gameObject.name istället för bara gameObject.name

Funkar nu.

Då ska vi bara ta bort objektet vi kollision.

Kallar på en function: Destroy(col.gameObject);

Enligt youtube video.

Detta funkar, mynten försvinner.

Lägga in så att countern uppdateras

Lägger in en ny text bit, sätter den bredvid Coins:

Gör den gul

Ger den ett appropriate namn

Söker Unity update text

Kommer in på answers.unity.com

Går tillbaka och väljer istället en youtube video: Change the text using script - unity 5.5 tutorial for beginner

Byter sökord: unity update text on collision

Kommer till answers.unity.com

Öppnar upp två stycken youtube videos från förra sökningen

Den ena samma som förut och är den som används vid uppdatering av scriptet

Men sparar den nya

Lägger in public Text text = count; ändras till lika med null

Och en till public int count = 0;

vid start läggs: text.coinCounter = count; ändras till text.text = count.ToString();

Lägger in using UnityEngine.UI

Fel i kod: debugging, ändringar finns i texten ovan.

Söker på converting int to string i c#

Hittade ToString();

Letar upp spelaren och hittar text i skriptets panel och lägger in text-objektet coin Counter

Lägger in count += 1 i funktionen som skapades innan för collision

Lägger också in text.text = count.ToString();

8.2.6. Iteration 6

Started with trying to find free audio snippets in an mp3 format (fund one on youtube and used an online converter to convert it to an mp3)

created an audio mixer and then deleted it

Imported the mp3 as a resource

google search "ADD SOUND EFFECT 2D"

looked at a online tutorial (studytonight)

thenne we renamed the mp3 to a easier name

the we open visual studio 2019 to edit our script

error misspelling

now we removed the checkered flag from the tile map and added it as a spirit

now we found a goal song and converted it to an mp3
added to son to the player object
and updated the script to play the song when colliding with the checkered flag
googled to try to figure out with sound will be played at the right time
attempted to create an array of the audio files didn't work as intended
Now we try using a youtube video to find audio.
I believe the sound worked
but there was no collision on the checkered flags.
checkered flag plays wrong audio
still issues with having multiple audios
a lot of google searches at this time and not funde a solution yet
now it works it was a misspelling error
now we added background music
worked finne. added a time stop when hitting the checkered flag to allow the music to play.

8.3. Further list of components

8.3.1. Godot Components

Rich Text Label

A node which can be used to display text and images and allow you to apply formatting to. We used this to display the amount of coins the player had picked up.

Signals

Signals are events, which nodes send when something happens to it, these signals can be accessed by writing a function in the script with the name of the signal, for example, body_entered which calls when the collision shape the script is attached to overlaps with another collision shape.

Collision layers and masks

Collision nodes in godot can be set to a layer, this is the layer which the collision node belongs to. They can also set a mask, which is all the other layers which the collision node can interact with. This way you can allow certain objects to collide and other items to not.

8.3.2. Unity Components

Camera

This object is used for viewing the world in the game, and is the way the player sees the game.

Canvas

A canvas is an object which holds all UI components which the camera will see. it was used to contain the text component for the coin counter.

EventSystem

A component which was created when adding the text component which the other team wasn’t sure what did and ended up not using. This component is used to send events to objects based on user input.

Tilemaps

Tilemaps are collisions of sprites which can be placed out in a predetermined pattern and have collisions applied to all sprites in the tilemap together.

layerMask

Raycasts in unity need to have a layer mask, which is a list of layers which the raycast will be able to see, By setting object layers to “Ground” you can tell the raycast to only see the ground layer and avoid the raycast hitting the player or other objects.

This was used to make the collision work properly and to make it so the player only can jump when it’s grounded (the layer), this was simply added to all the tiles in the tilemap.

boxCollider2D

This similar to the CollisionShape2D in godot, it gives an object a hit box that is used to detect if it collides with another object.

onCollisionEnter2D

An event which is triggered when the object enters the collision area of another object. it is used to decide what happens when two objects collide.

Audio Source

This is a component which can have a sound defined which can be played via code, similar to Godot’s AudioStreamPlayer2D.

getComponents<AudioSource>

This is a way to get a reference to a component on a GameObject, it is used on the audio source in the code to get all sounds added to the object, and returns an array of audio sources.