NTNU

Kunnskap for en bedre verden

Faculty of Information Technology and Electrical Engineering

PROG2900 - Bachelorthesis

Simulator for Lokførerskolen

Author:
Marco Ip
Julian Kragset
Michael-Angelo Karpowicz

19.05.2022

Summary of Graduate Project

Title:	Simulator for Lokførerskolen
Date:	19.05.2022
Participants:	Marco Ip Michael-Angelo Karpowicz Julian Kragset
Supervisor:	Aditya Suneel Sole, Tom Røise
Employer:	Isak Kvalvaag Torgersen, Norsk fagskole for lokomotivførere
Contact Person:	Marco Ip, piocram@gmail.com
Keywords:	Programming, Simulation, Train, Unity
Pages:	49
Attachments:	40
Availability:	Open

Abstract:

DeskSim is a train simulation game created in jMonkeyEngine by Lokførerskolen, where the objective is to teach students the principles of train conducting while giving them a safe environment to collect first-hand experience. We were tasked by Lokførerskolen with demonstrating the core functionalities of DeskSim in a new game engine. In this report, we will describe how we implemented core components of DeskSim in the Unity game engine, as well as detail the future steps that can finalize the project and implement features desired by Lokførerskolen. The purpose of this report is to detail what we have done to show that it is possible to recreate DeskSim in Unity.

Acknowledgement

This section is dedicated to the contributors who have helped us during the report and project. We would like to thank Aditya Suneel Sole for being our supervisor and giving us great feedback and support. Thank you to Isak Kvalvaag Torgersen for testing and feedback and for being a helpful client. Thanks to Emil Hjelle, Thomas Vincent Lien, Kristian Aakervik Rønning, and Amund Helgestad for providing good feedback and helping with the LaTex document.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Project Description

Our taskmasters for this project is, *Lokførerskolen* or better known as *Norsk fagskole for Lokomotivførere* (appendix C), and in this report we will refer to them as "train school" for writing purposes. The goal of the project is to select a modern *game engine* and prove that it is possible to recreate core parts of *DeskSim* in the new game engine. DeskSim is the name of the game simulation program that they developed and are currently using to teach new train drivers. This project consists of two main parts. The first part of the project consists of choosing a game engine, based on important requirements given by the train school, such as *Virtual Reality* compatibility, that the game engine is modern and will get long term support, and that it is compatible with the joysticks that the train school uses (*Saitek controllers*). The second and main part of the task is to develop a demo in the new engine, where we replicate some core components from DeskSim, to show that DeskSim can be recreated in our chosen game engine. Section 2 will delve further into the requirements given to us.

1.1.1 Background

The train school's purpose is to educate new train drivers and is located in Oslo, Norway. To prepare new train drivers, they make use of modern technologies like fully-fledged train simulation chambers, games, and VR technology, alongside regular classroom teaching for theory-based learning. Their simulator is called "DeskSim" and is a simulation game that lets the students get practical hands-on experience through scenario-based learning, all playable on their computers. It lets students put the theory into practice for a better understanding and learning experience. The simulation game is built on *jMonkeyEngine*, which is a game engine from 2003 [6]. Although it is still in use and maintained, the school is looking for new ways to improve their simulation. The reason for this is that jMonkeyEngine is an old game engine, which does not provide the same quality of tools or capabilities that other modern game engines do. For this reason, the school is starting to look for other alternatives to further develop into.

1.1.2 Goal

The goal of this project is to develop a simulation program that can be used as the basis for the next version of DeskSim, in order to better educate future train conductors at the train school. As we're developing this demo as the potential basis for a future simulation program, we want to make it easy to continue the development of this program, accommodating the implementation of other relevant mechanics, such as realistic railway coupling. The reason why we want to make it easy to develop in the future is that the school's objective is to modernize and improve their simulation game with more features than it currently has, so we should accommodate that. Development is done through our chosen engine, the *Unity* game engine, which we chose primarily because of their quality of documentation, the number of community resources, and the fact that it will be supported for a long time which guarantees that it is future proof. We want to ensure that this program is easy to use for the end-user and for the train school, who want there to be an easy way to create scenarios, so that many scenarios can be made to more efficiently separate and teach specific principles. To do this, we wanted to not only make a functional simulation that follows physical principles but also make it simple to use and edit scenarios, without compromising on Unity's already built-in world-editing capabilities. As students we also had an interest in, through this project, learning about things like terrain generation, procedural mesh generation, driver machine interface, mechanics of trains and signals, etc. The knowledge that we were creating something that has an actual real-world application, which would be the basis of the simulator that would educate many future train drivers, was a very important factor that made us hugely appreciative of our task and motivated to conduct this project.

1.2 Document Structure

The document is structured into ten chapters and at the end, you will find the appendices:

1. **Introduction:** Introduction to the thesis.
2. **Requirements:** Architecturally significant requirements and non-functional requirements.
3. **Development Process:** Describes the workflow during development and the tools used.
4. **Simulation Design:** Describes the initial design decisions made and alteration of the final product.
5. **Technical Design:** Describes system architecture, technical design, and game engine.
6. **Implementation:** Describes the implementation of features in the simulation.
7. **Deployment:** Describes the final build of the demo and its contents.
8. **Testing and User Feedback:** Describes usability testing process.
9. **Discussion:** Discusses the end results, utilized tools, and methods.
10. **Conclusion:** Describes our reflection on the project.
11. **Future Work:** Describes future work and improvement to the project.

1.2.1 Terminology

For an explanation of the terminology used in the report, please refer to appendix C.

2 Requirements

Our task given to us by the school consisted of mainly two parts. To see the full project task description, please refer to appendix A.

The first part of the task was to do an analysis of different game engines on the market and to write a report comparing at least three of them based on the requirements of the train school. In this report, we compared the following game engines; Unity, Unreal, Godot, and CryEngine. To see the full game engine report refer to appendix D. The game engines had to meet some set requirements, as per the client's request to use the project for further development. The game engine needs to be modern, it needs to be able to recreate elements from DeskSim, it needs to be suitable for VR development and it needs to be able to use a joystick as input for the simulator. After finishing the analysis, we were tasked to write a conclusion that would summarize what the group determined was the best choice as the game engine to use, and we chose to work with Unity.

The second part consisted of using the selected game engine to create a demo that would be similar to DeskSim. This also came with a set of rules as to what should be implemented, where the most important parts were the train, the rails, the signals, and the *Graphical User Interface*. For the full list of details, refer to appendix A. We were also given sub-goals if we were to complete the main assignment, which consisted of making developing tools related to creating *scenarios*.

2.1 Usability

One of the usability requirements that was not specified in the project task, but still important to the school, was that the simulator should use their Saitek controllers as input. This is to emulate the same levers that would be on board an actual train. As not everyone may have those Saitek controllers available, we decided to expand upon their requirement, by making other input devices able to control the simulator. The user can choose to use any input device recognized by Windows as a controller or keyboard, including the following:

- Joysticks / Saitek controllers
- Gamepad / Xbox / PlayStation controller
- Keyboard and mouse

2.2 Reliability

For a simulator or a game in general, it is important that it performs as intended and has minimal bugs. To minimize the chance of this and sudden errors we decided to set some requirements on how we develop the project.

One of the rules we made to improve reliability was to use one Unity version for the whole project. We chose Unity 2020.3.28f1 as our final version for development, as we did update our unity version once under development time, but tried to avoid any further updates.

Another way we mitigated sudden errors was how we used *GitHub* and *branches* to work. When a new branch was to be merged with the default *repository*, we made sure to review it before merging, ensuring that there were no flaws or errors in the branch.

2.3 Performance

For the performance of the simulator, there were no specific requirements given by the train school for how the simulation should perform, except it should run smoothly on the game engines. To do this, we tried to keep a high standard like allowing no simulation breaking bugs or glitches, writing code that utilized Unity’s built-in functionality as much as possible, allowing little to no crash occurrences, and ensuring that the joysticks the train school required to be compatible worked responsively and correctly.

3 Development Process

To organize and efficiently work during the development, we used a variety of tools and methods. In this section, we will discuss the use of said tools, as well as strategies and routines surrounding the development process.

3.1 Project Management Structure

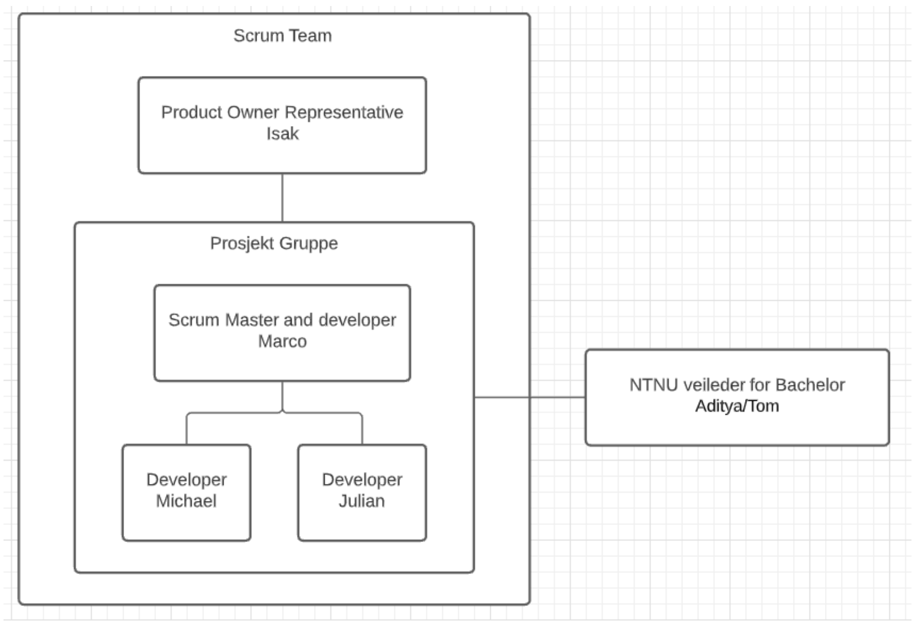

Scrum Team
Product Owner Representative Isak
Prosjekt Gruppe
Scrum Master and developer Marco
Developer Michael
Developer Julian
NTNU veileder for Bachelor Aditya/Tom

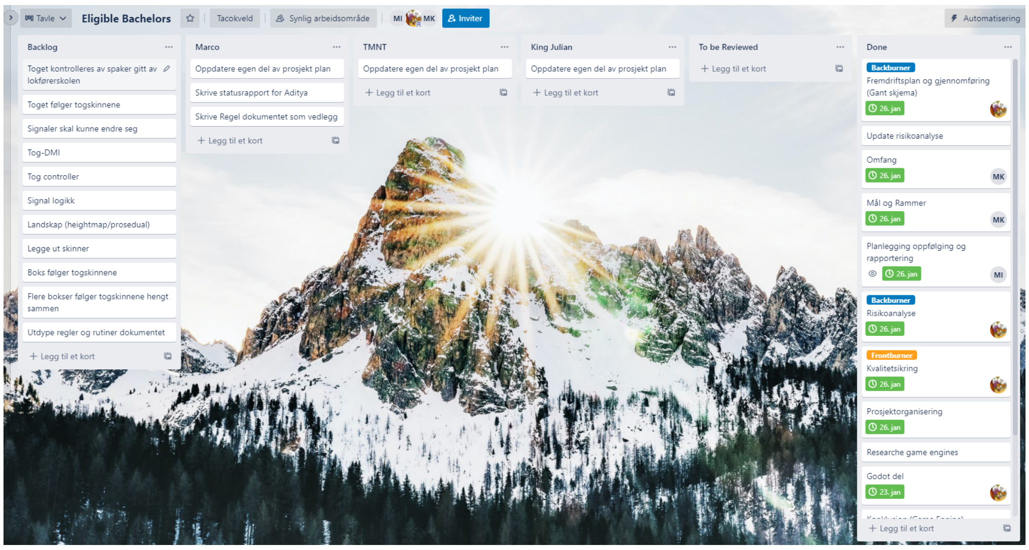
Figure 1: Organisation of the scrum team and members involved in the project

This is the basic structure of the project management (Figure 1) and the parts involved in the project. The people tied to the project consists of the group members, with Marco as the scrum master, Michael, and Julian as the project developers, Isak as the product owner representative, and Aditya as the NTNU supervisor. For the full project management structure and roles, please refer to appendix B, section 3.1.

3.2 Project Workflow

For our project workflow, we decided to go with the *scrum* methodology, as we had worked with this method before. This made the process easier as everyone was already familiar with it, and previous experiences had shown us that scrum is a good framework that suits the group's work habits.

Organizing our scrum process, we used a free web-based collaboration tool called Trello. We used Trello because we felt it gave us a good overview with its whiteboard-like structure and organization method. As well as being visually responsive, as the tasks moved from the product backlog to the done section (Figure 2). To represent the scrum methodology we made six columns, the *product backlog*, one for each of the members, to be reviewed and done. Tasks in the backlog column were the tasks that should be done to have the project completed. This was almost done already since we had to follow the requirements for the train school, however, we later added more tasks along the way, as more tasks appeared. The three-column for each of the group members represents the *sprint backlog*. We chose to divide the sprint backlog into three because we got a better overview of who was working on what and how many tasks members had. Then we have the review column, where tasks that had been completed would be reviewed at the next *sprint meeting*. Finally, we have the done column that would hold all the tasks that had been completed. This was to keep logs over who had done what, so that we could have every member held accountable for their assigned tasks and who was responsible for that particular task. We chose this way of structuring the project as it let us organize our tasks in a way that gave us a good overview of everyone's tasks, as well as providing a good overview of the project workflow as a whole.



Backlog
- Toget kontrolleres av spaker gitt av lokførerskolen
- Toget følger togskinnene
- Signaler skal kunne endre seg
- Tog-DMI
- Tog controller
- Signal logikk
- Landskap (heightmap/prosedual)
- Legge ut skinner
- Boks følger togskinnene
- Flere bokser følger togskinnene hengt sammen
- Utdype regler og rutiner dokumentet

Marco
- Oppdatere egen del av prosjekt plan
- Skrive statusrapport for Aditya
- Skrive Regel dokumentet som vedlegg

TMNT
- Oppdatere egen del av prosjekt plan

King Julian
- Oppdatere egen del av prosjekt plan

To be Reviewed

Done
- Backburner — Fremdriftsplan og gjennomføring (Gant skjema) — 26. jan
- Update risikoanalyse
- Omfang — 26. jan — MK
- Mål og Rammer — 26. jan — MK
- Planlegging oppfølging og rapportering — 26. jan — MI
- Backburner — Risikoanalyse — 26. jan
- Frontburner — Kvalitetsikring — 26. jan
- Prosjektorganisering — 26. jan
- Researche game engines
- Godot del — 23. jan

Figure 2: Scrum setup before development start

To effectively use scrum under development we set the *sprints* to be one week. This was to be able to follow the group members closely, while still giving us the freedom to work whenever they wanted. Every week, we would hold our scrum meeting at 12:00 on Monday, and go over what had changed on the Trello board.(Figure 2) Here we would first look at what was to be reviewed, and give feedback on whether or not it could be merged with the default branch.

If a task was deemed not good enough yet or incomplete, the task would be moved back to the person responsible for the task. Otherwise, it would be moved to the done column after being merged with the default branch, where it would be marked with the date that the task was reviewed and therefore completed.

After looking at the tasks to be reviewed, we would look at the sprint backlog. Here we would look at the individual member's tasks, and assign new tasks to the individual members if they were empty or close to done. After that, the meeting would be done, and the group would go over to a discussion and general work phase. In this phase, we could discuss other topics and work-related, to the project, and often work together online for a couple of hours.

During the project, we also had meetings with an NTNU supervisor, as well as a supervisor from the train school. Their job was to monitor the project and give feedback, one from an academic supervisor standpoint and one from a project supervisor standpoint. To make sure that we had good communication back and forth with them, we would have a meeting at least once every two weeks.

3.3 Development Workflow

Before we began to develop a new feature, we would go through some steps beforehand to ensure our effectiveness. First, we would discuss what features we wanted to add to reach our goals. Then someone would put those features as cards into Trello, with specifications like what functions were needed in this feature and a deadline. The person assigned to the card would have to complete it, and when it was ready, merge the feature into the develop branch. To do this, the group member would create a merge request on GitHub that later had to be reviewed and approved by someone other than the author. If there were any mishaps identified by the reviewer, they would comment on them. Then either the author or reviewer would repair it and commit their changes to the branch. It is important to mention that we also tried to lock the develop branch from any direct commits, but unfortunately, because our GitHub repository was private, it locked us from many settings that would usually be available for public repositories and required a GitHub Premium subscription to use them. The reason it would have been better to only allow merge requests was that every change would have to be reviewed and checked before they became final, but as long as we followed the same protocol this would not be too much of an issue.

3.4 Working Hours

Our working strategy consisted mainly of working alone and having online meetings every week. We chose this strategy because everyone was working remotely, meaning that the group members could only meet online. Another reason for choosing this strategy was to let the group member choose freely when to work, as our time schedules rarely lined up. To have this strategy work, we needed to have a set of rules, or else, it could lead to negative consequences. The most important rule was that everyone should work a minimum of 25 hours a week. For a better overview of the rules, look at appendix B, section B.1. For organizing our working hours and deadlines, we used a combination of Google Calendar for planning and getting an overview, as well as Clockify for time tracking and logging.

Google Calendar helped us to organize and plan for the coming events like deadlines and meetings. By using the calendar we could put events that were important for the group in a shared calendar, things like meetings with our bachelor supervisor, the train school representative, and our deadlines. The reason we used Google calendar is because of its sharing function, where you can share the calendar with other people and keep it synchronized. That way all of us would be able to see changes and get notifications on events that occurred, which helped a lot as we worked separately a lot of the time.

For tracking the team's overall and individual workflow we used Clockify. Clockify is a free time tracker software that let us track the working hours dedicated to the project [7] as well as serving as our time log. This lets us easily see if we are behind schedule or not and can help us manage our time better. We decided to use Clockify as it had the features we were looking for, time tracking and logging, and was easy to use. It also had a google extension that made logging hours easy, as well as compatibility with Google Calendar which let us see our events in the Clockify timesheet in an organized manner. To see the group's full-time log, look at appendix E.

3.5 Development Tools

3.5.1 Github

All group members were experienced with git through platforms like GitHub and Gitlab, so we chose GitHub as our version control system. We have worked together since we began our academic journey and were familiar with using git in many previous projects. Its functionality provided great support for us and made it easy to create and manage branches to work on, one function at a time. To host our remote repository has in effect worked relatively well. There were some issues sometimes when pushing changes where git did not catch all the changes done, which created some problems for us, but we managed to resolve these issues most of the time.

3.5.2 Visual Studio 2019/Code

For our most utilized *integrated development environments* or IDE, we used Visual Studio 2019 and Visual Studio Code for scripting and programming. Visual Studio offers packages from Unity that adds keywords from the Unity program that assists the developers and allows for a more efficient programming experience. We were most familiar with these and have experienced them to be reliable in the previous programs we have made. The Unity game engine automatically adapted to our chosen IDE and communicated to the IDE if there were any compiling or referencing errors.

3.5.3 Model converter

To be able to import some of the models that the train school gave us into Unity, we had to convert most of the model files externally. The models that had to be converted were .ac files, which the Unity editor could not import by default. To fix this we used a program called Blender, a 3D modeling software, to import the .ac models and exported them as .fbx models that Unity can import.

3.6 Assets and extensions

3.6.1 GLTFUtility

GLTFUtility is a free asset that allowed us, developers, to easily convert .gltf model files to .fbx without compromising anything. It automatically converted it as soon as the developer would paste the chosen .gltf file into the unity project, thereby skipping any typical extra steps through menus to convert it. It was not a necessary tool; we could have converted the files on another program manually or used any other external tools, but we used this since it is well made and easy to use.

3.6.2 MapMagic2

Our terrain is generated using MapMagic2, another free asset from unity's asset store. This asset is a set of tools that allowed us to generate terrain and texture it, by generating random Perlin noise and modifying it into something suitable for the terrain. It also lets us import heightmaps and generate from that instead of random noise.

3.6.3 VegetationDecorator

VegetationDecorator is a free extension that gave us more flexibility in the way we generate trees specifically, as we felt Unity's default method was lacking. It allowed us to generate better-looking forests, with many different kinds of trees, to replicate a more natural and realistic landscape.

4 Simulation Design

During the development of the project, we learned a lot about trains, however, we were not given any lessons on the theory behind train driving by the train school. We were not required to have any prior knowledge about trains, and therefore there might be errors in the technicalities tied to the train. All that we have learned is either from the train school representative answering questions or from individual research.

4.1 Initial Design

Our initial *domain model* (Figure 3) explains how we wanted to initially tackle and structure the major parts within the program. In the initial design, we wanted to make an event system that would control and be responsible for the signals, which would affect the rails, by making them open and close other paths. This would then be able to control where the train could go, and then trigger the event system bypassing triggers. To control the train we would use the train school's Saitek controllers as input, and pass it through an input interpreter to make the input easier to handle. As for the landscape, we figured out that we would implement a way to import heightmaps, which could be modified to accommodate the rails so that the terrain looks natural around the rails. As our first domain model (Figure 3), it worked as a good starting point for organizing and distributing work amongst ourselves.

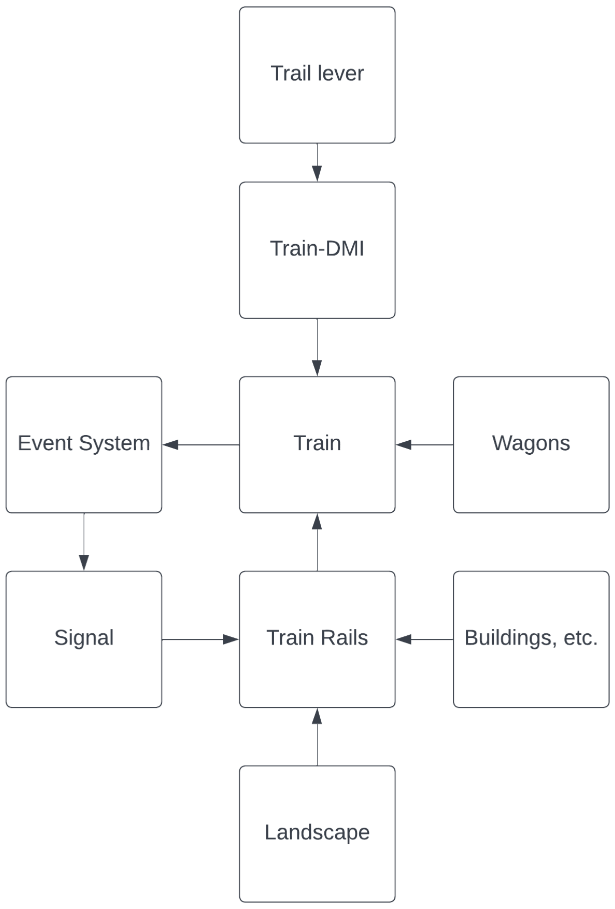


Figure 3: The initial domain model

4.2 Final Design

During the development of the separate parts in the initial domain model (Figure 3) we encountered difficulties that would lead to us having to redesign the model. After a lot of redesigning and finalizing the project, the final domain model became this (Figure 4)

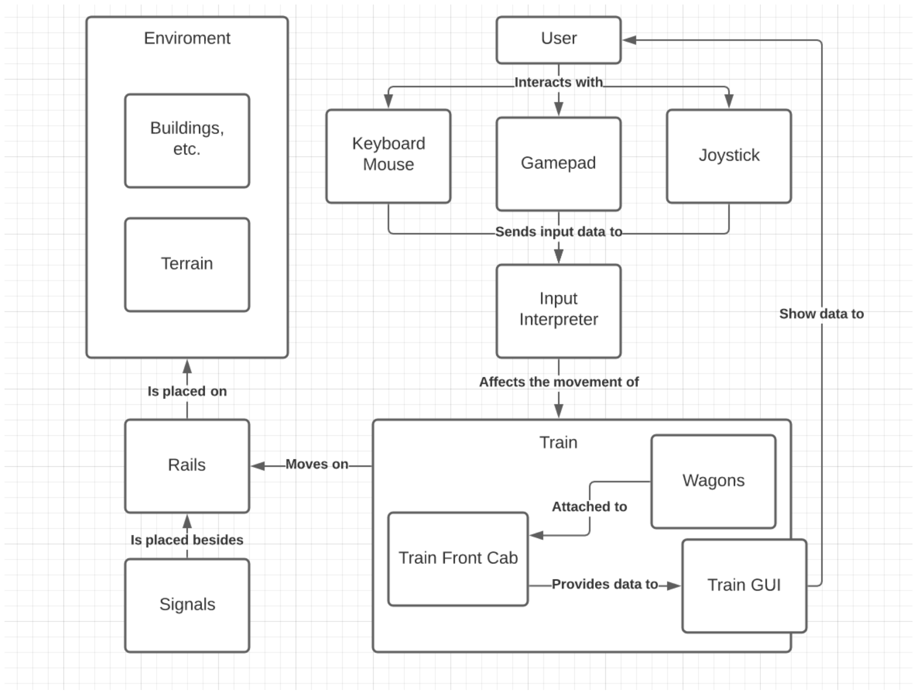


Figure 4: Final domain model

Here are some architecture models for the individual parts in the final design.

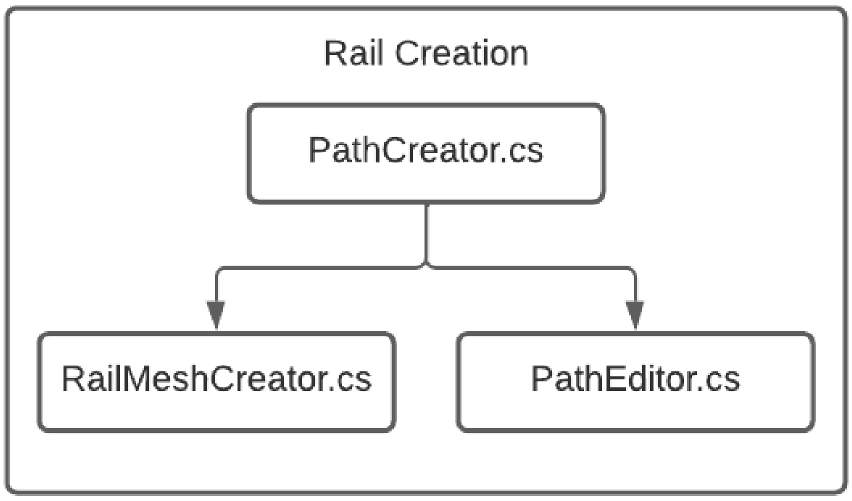


Figure 5: Rail architecture model

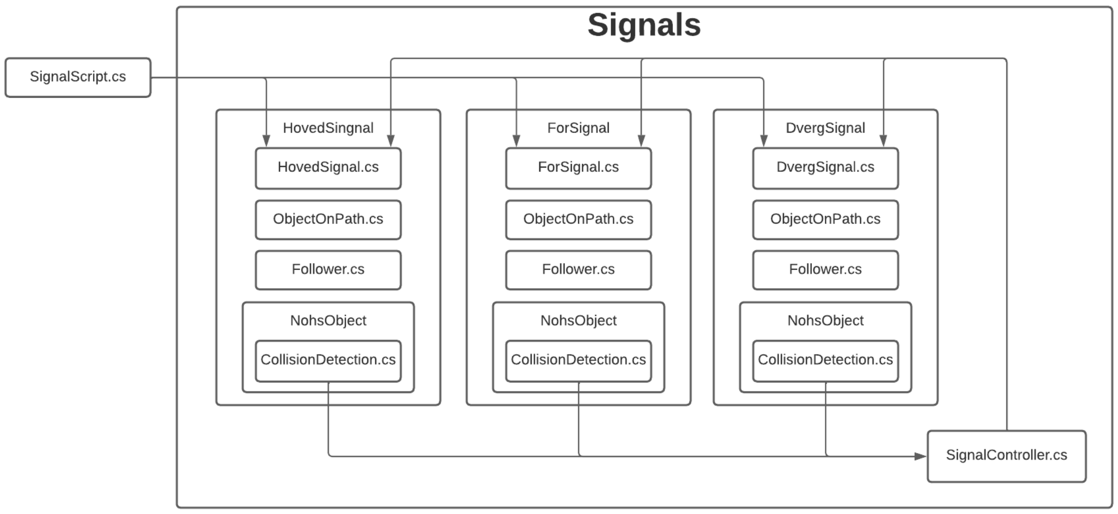


Figure 6: Signal architecture model

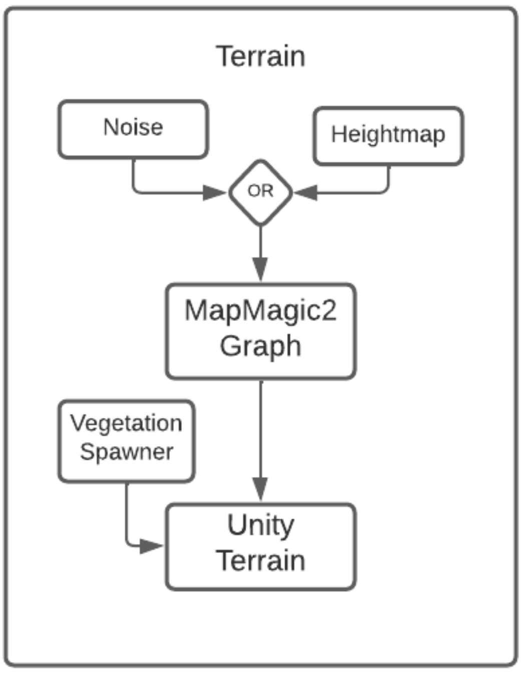


Figure 7: Terrain architecture model

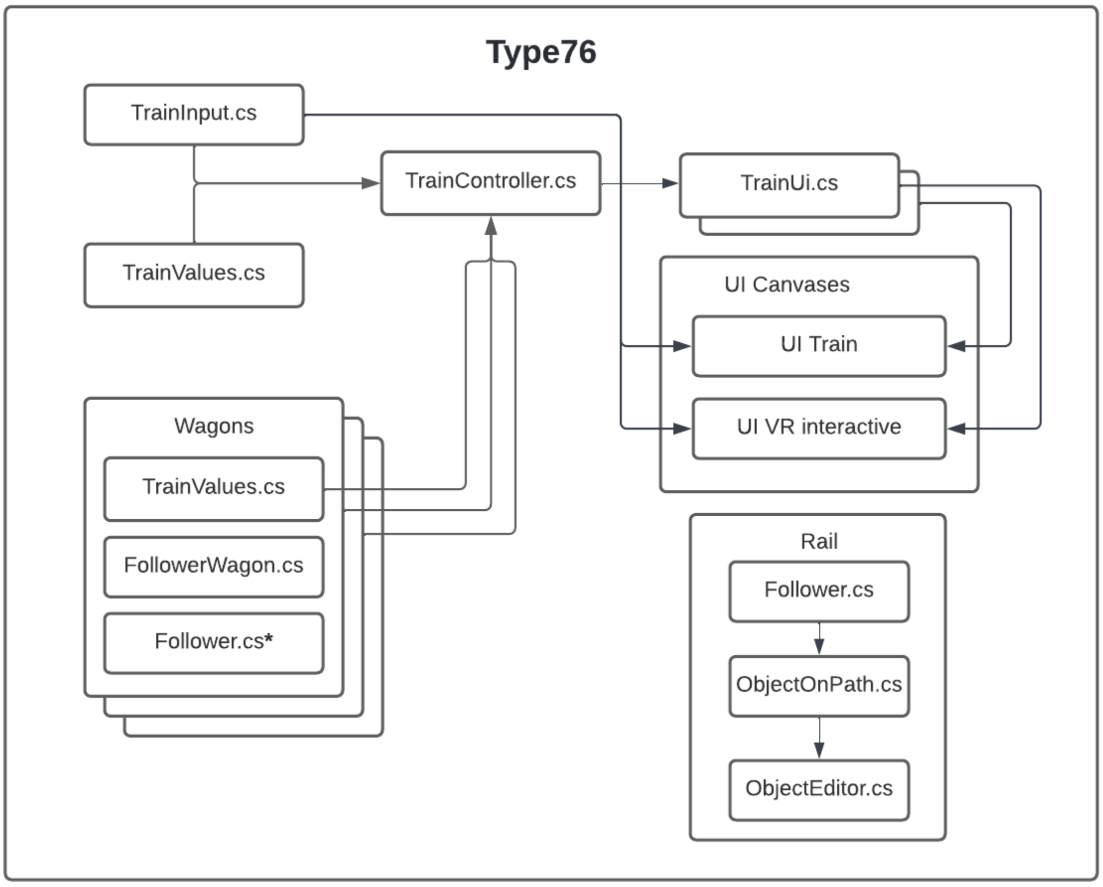


Figure 8: Train architecture model

5 Technical Design

5.1 Unity Engine

For information about the Unity game engine, refer to the game engine report in appendix D, section 2.

5.2 Train model

The train is one of the main parts of the simulator, and it is through the train, that the user can interact with the environment. The train prefab consists of multiple parts and models that together make up the train as the model shows (Figure 8).

The train is built with the models provided by the train school and contains additions that we have added onto the *prefab*. One element is the *GUI* which can be divided into multiple parts. (Figure 9) The gauges tell the user about the speed of the train in kilometers per hour and the pressure on the brakes, in bar. The two GUI levers on the side are an optional way to control the speed and braking of the train, by using the mouse to interact with them. This is to make the simulator more flexible for people who do not have a lever-type controller available. If the user does have a lever controller, the GUI levers will mimic the position of the physical levers. The last element of the GUI is the reverse switch button that is responsible for changing the direction of the train, either forwards or backward. The button will switch to a red button if the reverse is engaged, and green if the train is driving forwards.

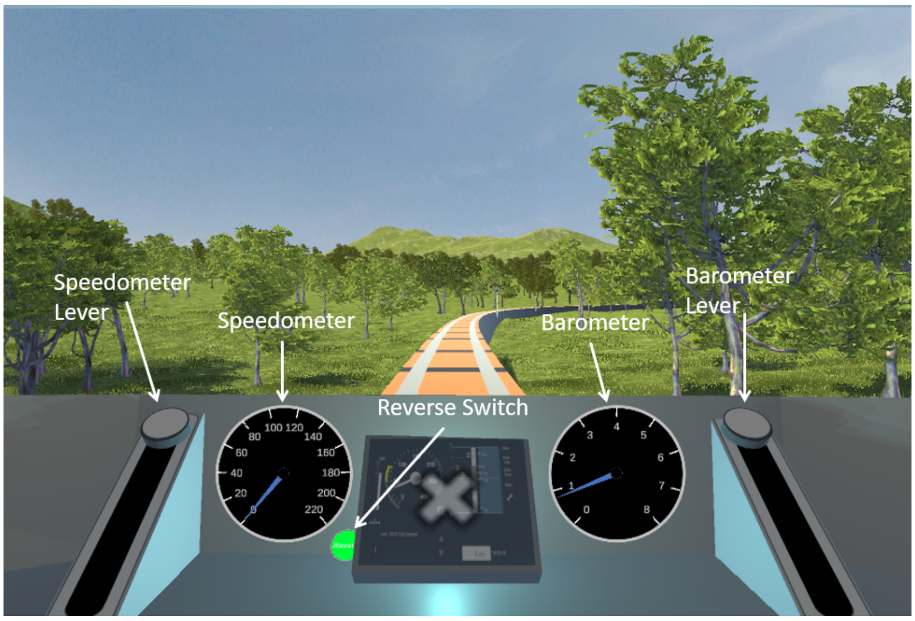


Figure 9: The train GUI in the simulator

To get an accurate feeling and simulation of the train, as requested by the train school, we used real-life data and values from the cite "Norske tog" [5]. The train is supposed to model the train "Type76" however we could not find the specifications on that train model, therefore we opted to use the train values for Type75. This includes the pulling and breaking force of the train, top speed, and weight.

5.3 Physics

The physics of the train is one of the most important parts of the simulation, as it gives the train a realistic feeling, which is important for creating a good learning environment. At the same time, physics can not be too complex, as this will make the solution more complex and resource-intensive.

When looking at a real scenario, a train will be affected by a lot of different forces. Forces work on all objects around us and allow them to stay in place and move around. To give the simulation a realistic feeling while still keeping it relatively simple, we decided to remove all friction. That means that the train will not be affected by forces that can be unpredictable and immeasurable, and consists of mainly air resistance and friction from the rails. This makes the process of calculating the acceleration of the train easy, as we just use Newton's second law $F = m * a$ [3] to find the acceleration we should apply to the train.

5.4 Signals

Signals were also developed according to the requirements set in the task description, and we were given three models to implement into the demo. In Norwegian these are called "Hovedsignal", "forsignal" and "dvergsignal", but for writing, we are going to refer to them as main, front, and dwarf signal.

On each of the signals, you can select the signal starting light pattern and what light pattern to change to after a condition is met. As a proof of concept, only a timer and instant switching of light pattern has been applied. After all of the lights are programmed in the editor, the signal controller that all the signal objects are children off would then be responsible for changing the lights from the starting to the outcome signal light pattern.

5.5 Input

Inputs were handled by Unity's new Input System module, which is a more modular and dynamic way to handle input than standard Unity input. This makes the system more flexible as it now supports cross-platform controls and allows us to use the Saitek Controllers as well as other inputs for the system. This system essentially works by providing an *InputActionAsset object*, in which you can create general *actions* (e.g. *move* or *jump*) and tie many different inputs ("bindings") to the same action. These actions can then be organized into "action maps", like having one action map for the train and one for the player. This makes it easy to change between these control schemes and works by disabling train controls when exiting the train and enabling player controls to walk around. Each action can be tied to Unity *events* or be broadcast as C# events, which is what we're doing. This InputActionAsset object then compiles a C# class from our configuration, which is what our scripts use to read input. This lets us detect when actions are triggered or changed. So when one of the bindings is triggered, like when a joystick changes its value, a C# event is called. Which any script can pick up on and read the value from.

Inputs were organized by creating dedicated scripts for each action map. Each action map controls the inputs by binding them from the action map to their respective C# events, which are the TrainInput and PlayerInput class (Figure 10). TrainInput ties the button-presses to different functions and values, and handles reading the raw input from the joysticks and converting it to a zero to one value. It also sees if the user doesn't have a compatible joystick attached, lets them use buttons or controllers instead, and also applies modifiers that can be set in the settings like inverting the sticks. It also has a function to synchronize GUI sliders with the pressure or acceleration, allowing the user to use their mouse to control these things if they want to. Other scripts can then read the values from TrainInput to use them, such as TrainController which controls the train physics, and TrainUI which renders the GUI gauges. This is also true for PlayerInput and PlayerController, though these aren't used in any scenarios despite being mostly implemented.

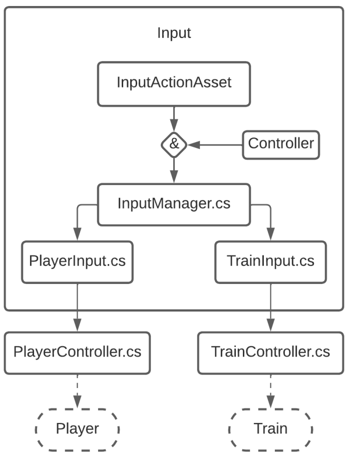


Figure 10: Input graph

5.6 Menus

The requirements we recognized for our menu system were quite simple. The application required an interface that would allow the user to change the settings for key-binds, choosing scenarios, and an in-game menu that returns the user to the main menu. We implemented this by using standard Unity GUI components, using the functionality to group objects, and disabling and enabling them to switch between the screens, like the scenario loading and main menu screen. Below is the structure of our main menu (Figure 11) and the screen for loading scenarios (Figure 12).

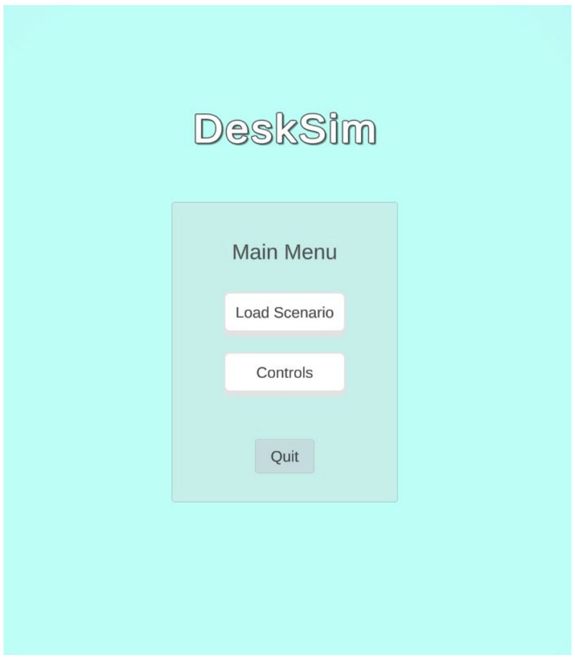
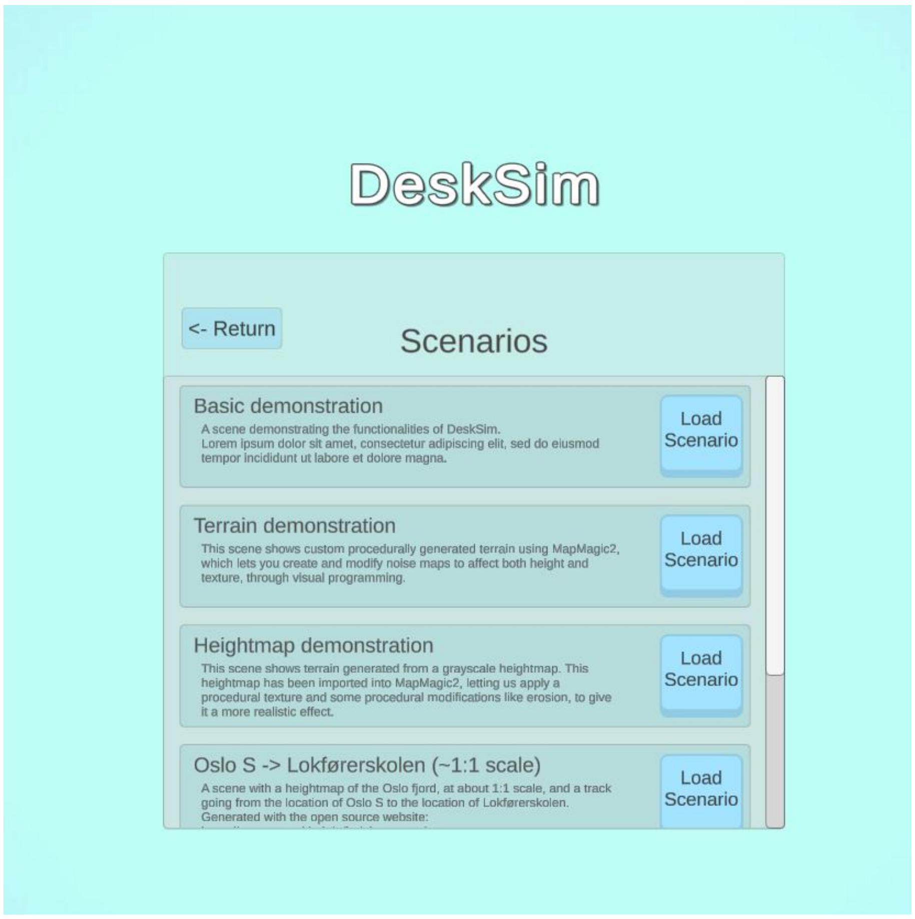


Figure 11: Main menu



Basic demonstration

A scene demonstrating the functionalities of DeskSim.
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna.

Terrain demonstration

This scene shows custom procedurally generated terrain using MapMagic2, which lets you create and modify noise maps to affect both height and texture, through visual programming.

Heightmap demonstration

This scene shows terrain generated from a grayscale heightmap. This heightmap has been imported into MapMagic2, letting us apply a procedural texture and some procedural modifications like erosion, to give it a more realistic effect.

Oslo S -> Lokførerskolen (~1:1 scale)

A scene with a heightmap of the Oslo fjord, at about 1:1 scale, and a track going from the location of Oslo S to the location of Lokførerskolen.
Generated with the open source website:

Figure 12: Main menu after the user has clicked "Load Scenario"

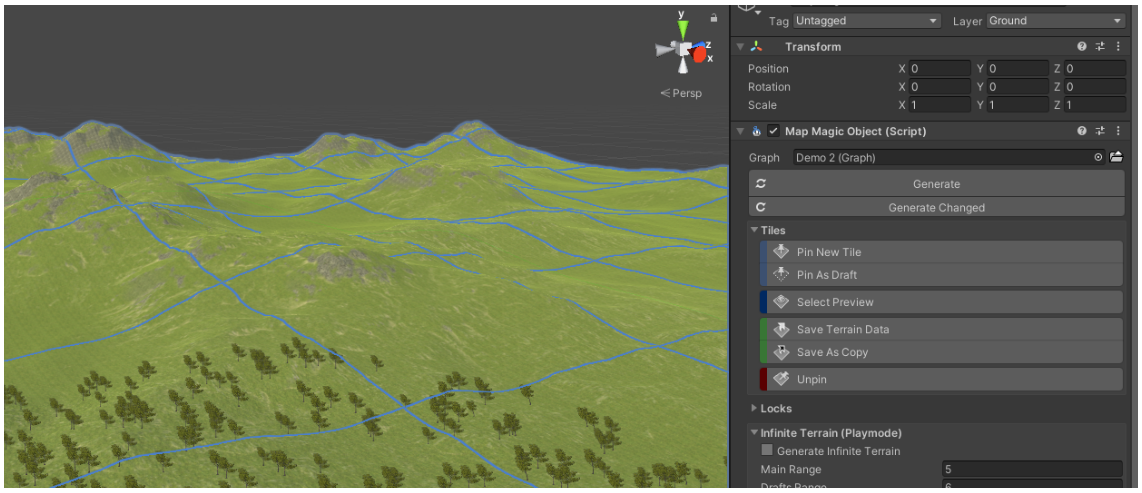
5.7 Terrain

The basic requirement for the terrain was to be able to shape it manually with terrain tools and texture it and place objects and rails on it. However, during conversations with the train school, there were also expressed interests in more complex terrain features, such as generating terrain from *heightmaps* and procedurally generating terrain, as long as the terrain could still be manually shaped to suit the steepness for the train tracks.

The basic requirement of a shapeable terrain was met by using Unity's default terrain editing tools, however, this was very limited and didn't allow for texturing the terrain without having to write custom *shaders*. This would make it difficult for a person inexperienced in graphical programming, to make new scenarios for the simulator. This is important to us because one of the school's desires and extended goals was for scenarios to be easily created by someone without much programming and technical experience. So we kept this in mind and looked for a better implementation of our terrain.

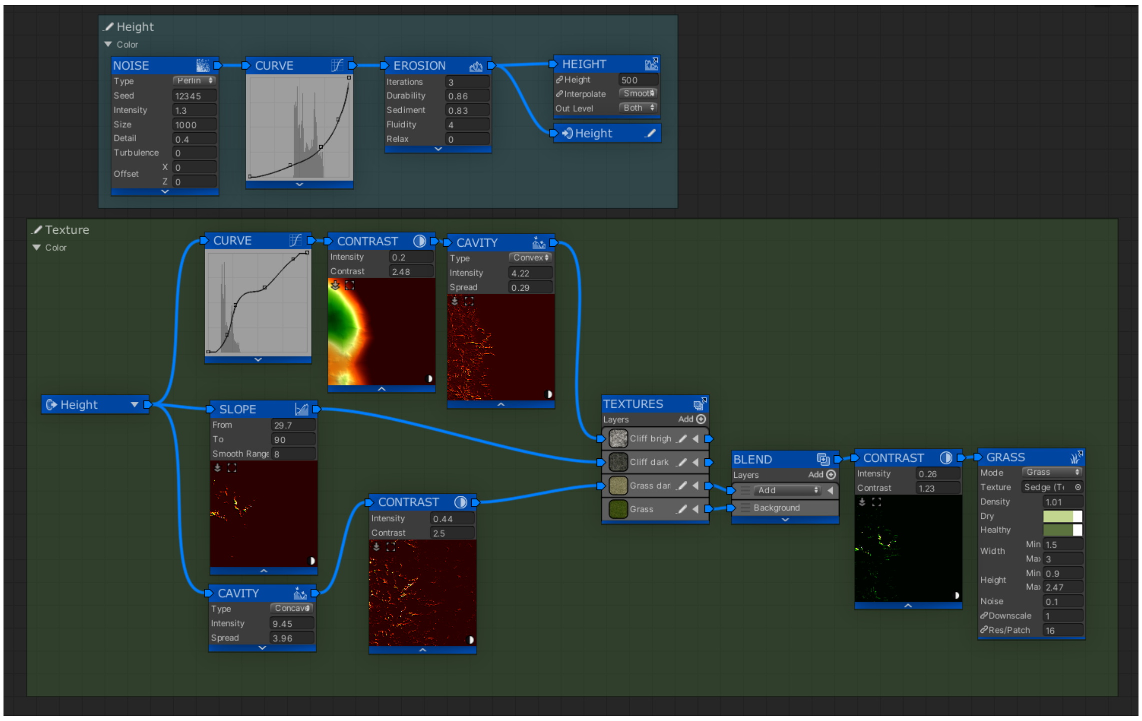
The best way we eventually found to get the desired features within our budget and time frame was through the free Unity asset MapMagic2. This asset uses the standard Unity asset license, which poses no problem for the uses the train school is interested in. It also provides plenty of features that are excellent for our use, while offering paid extensions to allow for more complex features that might also be interesting in the future, like procedural object placement. However, we currently use Unity's standard object placement tools for this.

MapMagic2 allows us to procedurally generate both terrain and texture using *noise* or Perlin noise [1] as a generation method. This method consists of pseudo-randomness and plenty of noise modifiers that we can string together to create just the kind of terrain we desire. This tool-set lets somebody with some knowledge of how noise works, really quickly and intuitively create procedural texturing and procedural terrain, which, with some noise modifiers, can be made suitable for placing tracks on without having to manually modify the terrain. An example of this can be seen in Demo 2 (Figure 13), which features completely procedural terrain that hasn't been manually modified in any way, instead modifications were applied to random noise to create suitably flat terrain. The graph creating the terrain can be seen in figure 14.



Tag Untagged Layer Ground
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Graph Demo 2 (Graph)
Generate Infinite Terrain
Main Range 5

Figure 13: Demo 2 terrain as seen in the editor



NOISE: Type Perlin, Seed 12345, Intensity 1.3, Size 1000, Detail 0.4, Turbulence 0, Offset X 0, Z 0
EROSION: Iterations 3, Durability 0.86, Sediment 0.83, Fluidity 4, Relax 0
HEIGHT: Height 500, Interpolate Smooth, Out Level Both
CONTRAST: Intensity 0.2, Contrast 2.48
CAVITY: Type Convex, Intensity 4.22, Spread 0.29
SLOPE: From 29.7, To 90, Smooth Range 8
CAVITY: Type Concave, Intensity 9.45, Spread 3.96
CONTRAST: Intensity 0.44, Contrast 2.5
TEXTURES: Layers Add
BLEND: Layers Add, Add
CONTRAST: Intensity 0.26, Contrast 1.23
GRASS: Mode Grass, Texture Sedge, Density 1.01, Dry, Healthy, Width Min 1.5 Max 3, Height Min 0.9 Max 2.47, Noise 0.1, Downscale 1, Res/Patch 16

Figure 14: The graph that generates the terrain and textures in Demo 2

MapMagic2 lets us use the same tools but on heightmaps instead, the following pictures are of Demo 4 (Figure 15), which is a 1:1 scale recreation of a portion of the Oslo fjord, and the heightmap it is generated from (Figure 16).

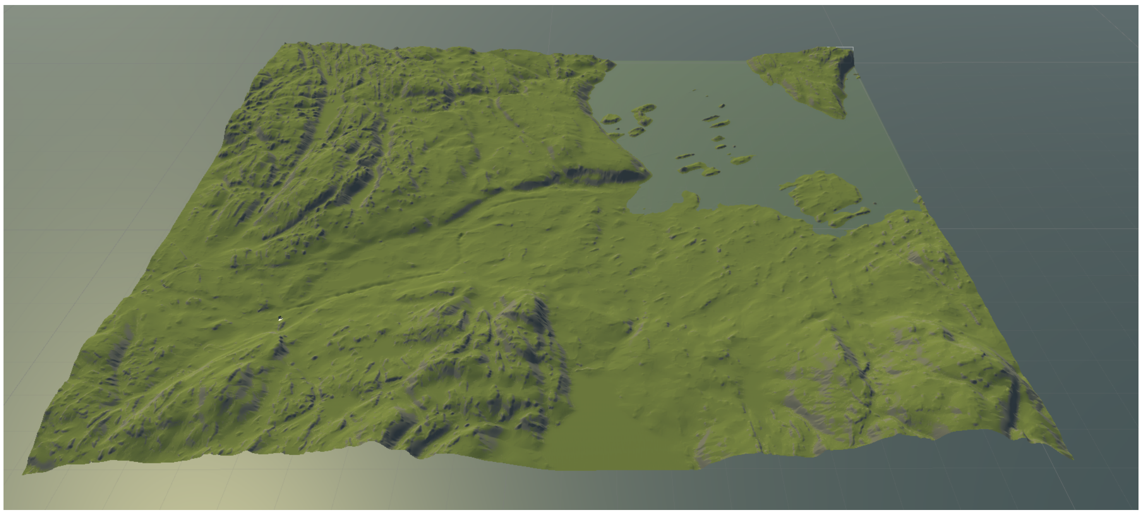
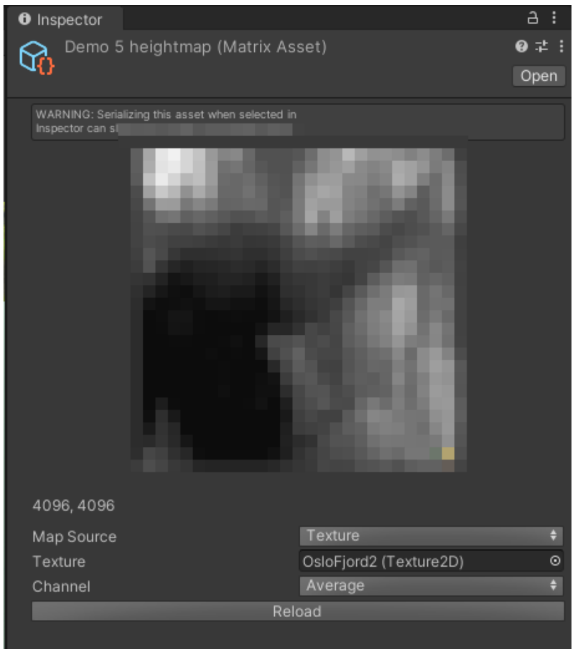


Figure 15: A 1:1 scale map of Oslofjorden, generated from a heightmap with MapMagic2. Sources listed below



WARNING: Serializing this asset when selected in Inspector can sl

4096, 4096

Map Source: Texture
Texture: OsloFjord2 (Texture2D)
Channel: Average

Figure 16: The heightmap used to generate the terrain of Oslofjorden. Blurred for licensing reasons because the data itself isn't necessary to show.

This heightmap of the Oslo Fjord is sourced from the Tangram Heightmapper, which itself sources data from Mapzen's Elevation Service, which aggregates multiple sources of data. These sources are checked and allow free access to their data, but wishes that they be credited. 30 meter data is sourced from SRTM and bathymetry is sourced from ETOPO1 for which NCEI is the source. SRTM requests that the following statements be made, data from USGS Products is available from the U.S. Geological Survey, and NASA Land Processes Distributed Active Archive Center (LP DAAC) data are distributed by the Land Processes Distributed Active Archive Center (LP DAAC), located at USGS/EROS, Sioux Falls, SD. http://lpdaac.usgs.gov [2].

The most important point for the train school was that they could manually edit the terrain to accommodate the tracks. That is something that we have taken into account as MapMagic2 generates its terrain using standard Unity terrain underneath. Therefore any terrain generated can be manually edited using Unity's powerful built-in terrain tools. This also solved another big problem we had, that terrain generated using height-maps with Unity's default tools, has the same fidelity as the resolution of the height-map. This meant that any height-map imported would have a very low resolution and so they couldn't be shaped to accommodate trains because the low fidelity terrain would always lead to rough terrain with jagged edges. The use of default Unity terrain underneath also lets us use a free third-party vegetation spawner Asset, "Vegetation Spawner" [8], allowing for more complex vegetation generation than standard Unity allows.

Here is a model showing the structure of our terrain, figure (Figure 17)

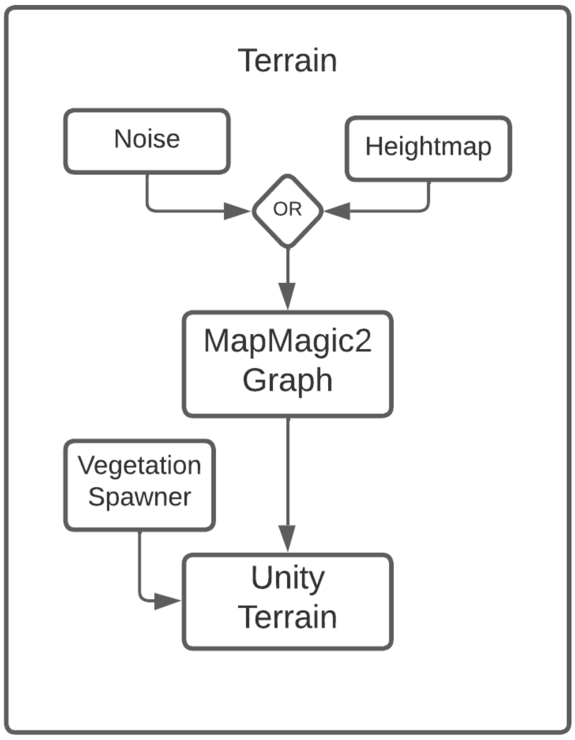


Figure 17: Terrain graph

These are the tools we ended up using for designing our terrain, and how they cover all the needs of the train school and the issues we foresaw.

5.8 Scene Management

Most of the scenes in the scenario folder are meant for demonstrating specific mechanics we implemented. Other scenes are served as a testing playground for us to mostly experiment, test, and identify problems. Demo scenes usually contain terrains with varying parameters like size, height thresholds, etc. They also contain predetermined paths for rails and include a train on them. Some of the later scenes include the signal we implemented since it is one of the later features we finished developing.

In-play mode, you can choose different scenes to run once you pressed the play button in the main menu. The purpose is to easily manage and choose what scenarios the user wants to run. The train school has something similar to this functionality. We wanted to capture and improve it by having the runtime option, instead of stopping the program completely, choosing the next scenario on another window, and starting the booting of the program repeatedly.

5.9 Rails

In the initial design, we had to think about how we were going to design the rails, to satisfy the requirements received by the train school. The rail object *mesh*, so that the train could easily navigate the rails when the program was playing. To make developing scenarios easier, we designed the rails so that 3D objects like train cabs, wagons, and signals could snap onto the rails. Not only would these objects have to move on the rails, as if they followed them, but also rotate according to the defined *normals* on the rails. Users would then get the impression that the objects would stay attached to the rails and it would make it easier to set up scenarios. The other crucial condition we had to think about was that rails had to be easy to create, edit, and erase. That meant we would have to try to avoid using pre-modeled rail models, especially when it could with ease conflict and clip through the terrain mesh that way. We were also required to allow for the rail to be curved so that maneuvering and driving in different directions could be possible.

Developing and managing all of this would have been considerably time-consuming. Therefore we had to be creative and find a solution that could help us make the rails fulfill our requirements whilst also making them easy to use and maintain. *Bézier curves* appeared as a great answer to our problem. It involves a set of anchor points that serve to define smooth and continuous curves along with those points. In the Unity Asset Store, there are assets available for free that allow for the creation of Bézier curves. The asset we used is called Bézier Path Creator[9], which is a small development tool that allows for the easy creation and modification of three-dimensional Bézier curves. It includes easy-to-manage editor tools that worked well on our unity version. The Bézier curves created with this asset have many practical modifiable parameters that fit our objective, which can be changed in the *inspector* of the curves in editor mode, allowing for things like freely editing the anchor point's position to define a suitable path. There are automatic smooth curvatures between points, but the extension also allows us to perform manual curve modifications in case we require more precision. There is also an option to perform global modifications on the normals of the path, letting us specify the direction of all normal angles on the path, which decides which way the rails and train are oriented. In the display options for the path, the way the path and all its components are present visually can be changed, for better visibility in the editor. This is helpful in an event where the developers zoom out far enough they cannot see the anchor points anymore, so they can change the size of the points to easily identify them better.

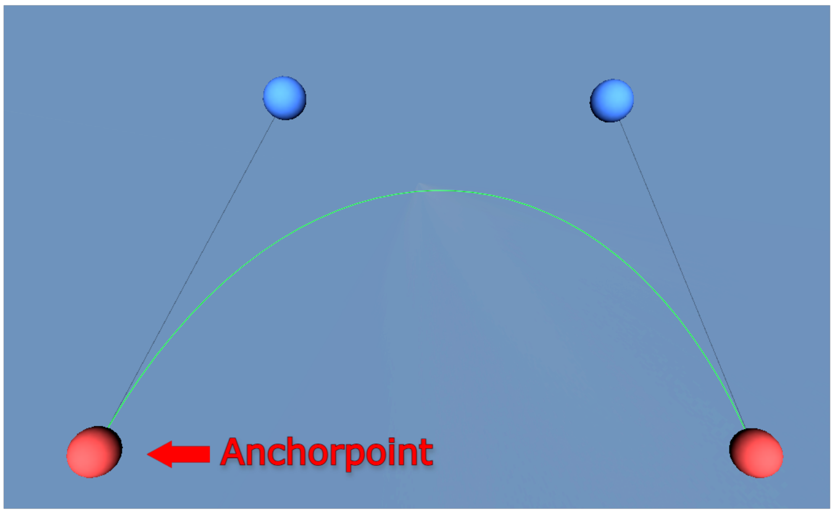


Figure 18: Anchorpoints of the Bézier Path Creator

Although this tool has great potential and helped us make this project better, it did have some issues and limitations that we had to deal with. The report will elaborate on these issues further in section 9.

6 Implementation

In this section, we will go over the core implementations of all the different parts of the simulation.

6.1 Train physics and controls

To implement the train physics, we use the "Rigidbody" component given to the train object, which is a default component in Unity that allows for the use of built-in physic calculations. This allows us to move the train object in the scene by setting variables like position, velocity, and acceleration on each update frame. To move the train forward we use the formula $F = m * a$ to find the acceleration to give the train, as we know both the mass and the pulling force of the train. However, we also have to consider the position of the acceleration lever as the conductor of the train decides how fast the train should go. To solve this we multiply the normalized value of the lever, by the pulling force, and get the acceleration based on the acceleration controller. See the code down below.

This is the function that handles the acceleration for the train.

```
From TrainController.cs
/**
 * Calculates the acceleration force to be applied on the train
 *
 * @return                      Returns the acceleration force
 */
private float GetAccelerationForce()
{
    return tValues.MaxPullingForce * input.acceleration / (tValues.Mass +
    ↪  totalWagonMass);
}
```

The brakes on a train operate on pressure. That means that if the barometer is zero the train will not move, because the brakes are engaged. To make the train move, we have to loosen the brakes by pressurizing them to 5 bar, which is controlled using the pressure lever. This means that the pressure lever does not directly control the brakes, but controls the amount of pressure that is let into the chamber that controls the brakes. To mimic this behavior, we let the lever control the pressure and depressurization of the brakes while the braking force is calculated from how pressurized the breaks are. See the code down below.

The first function computes the breaking acceleration and the other function is responsible for increasing and decreasing the pressure.

```
From TrainController.cs
/**
 * Calculates the force needed to stop the train as kinetic energy over time
 *
 * @return                      Returns the break force
 */
private float GetBreakForce()
{
    return - ((tValues.MaxPullingForce + totalWagonBreakForce) * (1.0f -
    ↪  (pressure / 5.0f))) / (tValues.Mass + totalWagonMass);
}

/**
 * Adds or subtracts pressurevalues based on the input.
 */
```

```
private void UpdatePressure()
{
    if (input.pressure <= 0 && pressure >= 0)
    {
        pressure -= 0.007f;
    }
    else if (input.pressure > 0 && pressure < 5.0f)
    {
        pressure += 0.005f * input.pressure;
    }
}
```

6.2 Signals

The signals are implemented in such a way that more signals can easily be made using a lot of the same functions. To make this work we have used inheritance. The signal's individual script has to derive from the SignalScript class, which contains functionality that all the signals share, to minimize duplicating code. This individual script is responsible for displaying different light patterns and other behavior, like blinking in the main signal. This functionality is *hardcoded* so that it can reliably display the correct light pattern.

The SignalController class sits on the parent object of all the signals and is responsible for changing the individual signal's light pattern. To detect the train in the simulator for changing the signals we use collision boxes both on the signal and on the train, where the train is marked with the tag "Train".

This is the function that handles the collision detection for the signal.

```
From CollisionDetection.cs
/**
 * The function handles detecting the train when interacting with the signal
 */
private void OnTriggerEnter(Collider collision)
{
    if (collision.tag == "Train")
    {
        transform.parent.GetComponent<SignalScript>().CollisionDetcted(this);
    }
}
```

If the signal has detected the train, the signal will switch a boolean statement, causing the SignalController to detect the boolean change, and change the light pattern of the signal based on the preset from the editor. To make this code more readable we use enums to represent the input for the different light patterns, which can be found in the SignalEnum class. It is also the SignalController class that allows the front signal to connect with the main signal behind it as the functionality of the front signal is to preview to the conductor what is up ahead.

This is the function runs every frame and detects the signals when they are triggered by the train.

```
From SignalController.cs
private void Update()
{
    // Checks if one of the lights in has been triggered
    for (int i = 0; i < listOfSignals.Count; i++)
    {
        int type = listOfSignalType[i];

        Transform signal = listOfSignals[i];

        // If the signal has a detection mode then test if the appropriate signal has been triggered
        if (signal.GetComponent<SignalScript>().TrainDetectionType != DetectTrain.Ingen)
        {
            switch (type)
            {
                case 0:
                    if (signal.GetComponent<DvergScript>().TrainTrigger)
                    {
                        SenarioManager(signal, type);
                        signal.GetComponent<DvergScript>().TrainTrigger = false;
                    }
                    break;
                case 1:
                    if (signal.GetComponent<ForSignalScript>().TrainTrigger)
                    {
                        SenarioManager(signal, type);
                        signal.GetComponent<ForSignalScript>().TrainTrigger = false;
                    }
                    break;
                case 2:
                    if (signal.GetComponent<HovedSignalScript>().TrainTrigger)
                    {
                        SenarioManager(signal, type, i - 1);
                        signal.GetComponent<HovedSignalScript>().TrainTrigger = false;
                    }
                    break;
                default: Debug.LogError("Not a valid sign number: " + type); break;
            }
        }
    }
}
```

6.3 Rails

The rails are one of the most essential and intricate parts of this project. Rails are able to hold objects on them by attaching them to the rail path. They also hold the data for the rail path normals, which are perpendicular to each point on the path with specified intervals.

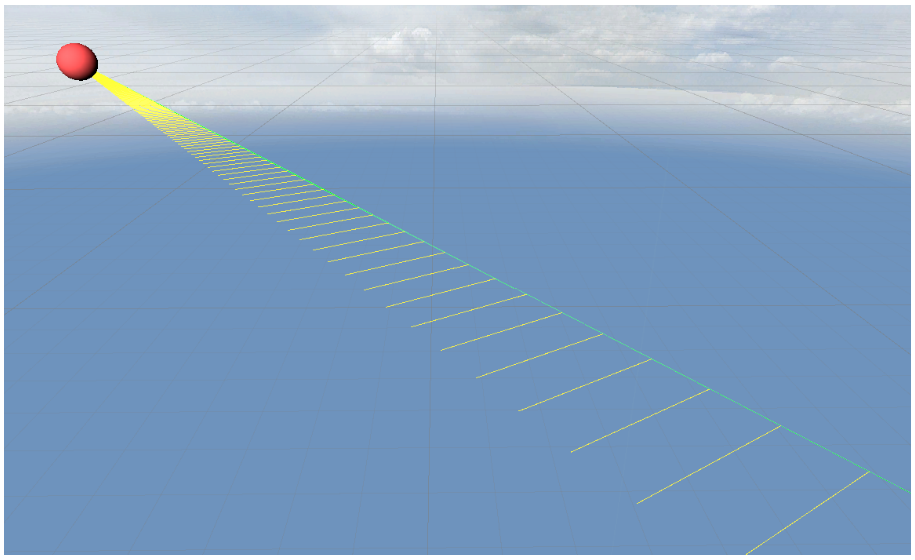


Figure 19: Visualization of normal lines (Yellow) on path line (Green) in Unity Editor

Normal data is used to rotate attached objects based on the path. The path itself can direct objects, in our case, the train and its wagons which will be able to follow the pre-constructed path in play mode. In editor mode, we can place objects like signals and the starting position of the train.

6.3.1 Bézier Path

Once you added the PathCreator script onto a game object, you will be able to identify a new object in the editor view that appears to be two points with a curved line in between.

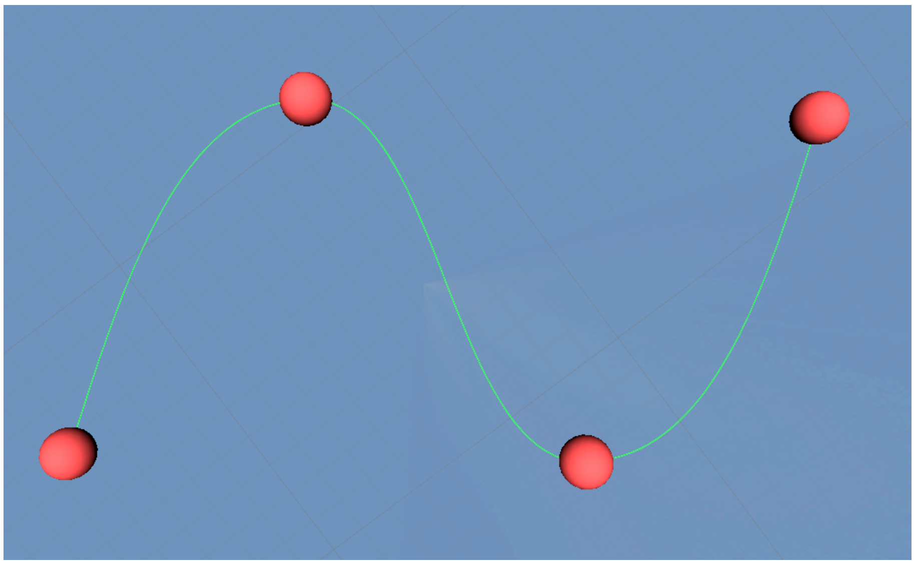


Figure 20: Bézier Path Creator editor in automatic mode

The automatic mode automatically makes smooth Bézier curves between anchor points.

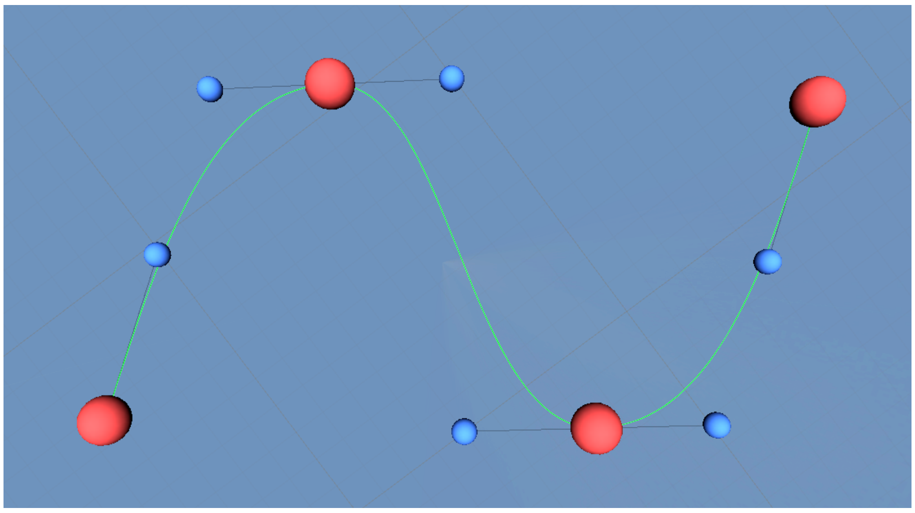


Figure 21: Bézier Path Creator editor in free mode

In free mode, you can define the curves more thoroughly which requires manual editing. The Bézier Path Creator, as its name reveals, is based on the Bézier curve algorithm, which is a parametric curve used often in graphical computation processes. Usually, a set of control points will define a smooth and continuous curve to approximate real-world shape. It is used in the bodywork of cars, font designs, and animation.

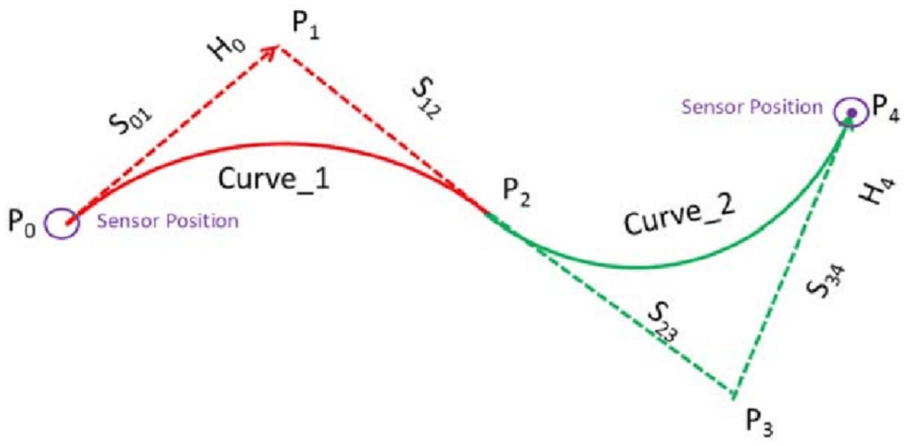


Figure 22: The components of a 2D cubic bézier curve

In this project, it is used to draw a fine smooth curve between lines that you can manually edit. For this asset to suit our needs, we needed to make some additional scripts and changes to change its behavior. One handcrafted change that bore importance for the editing part of rails, was adding a *raycasts* from the mouse position to project onto the terrain mesh.

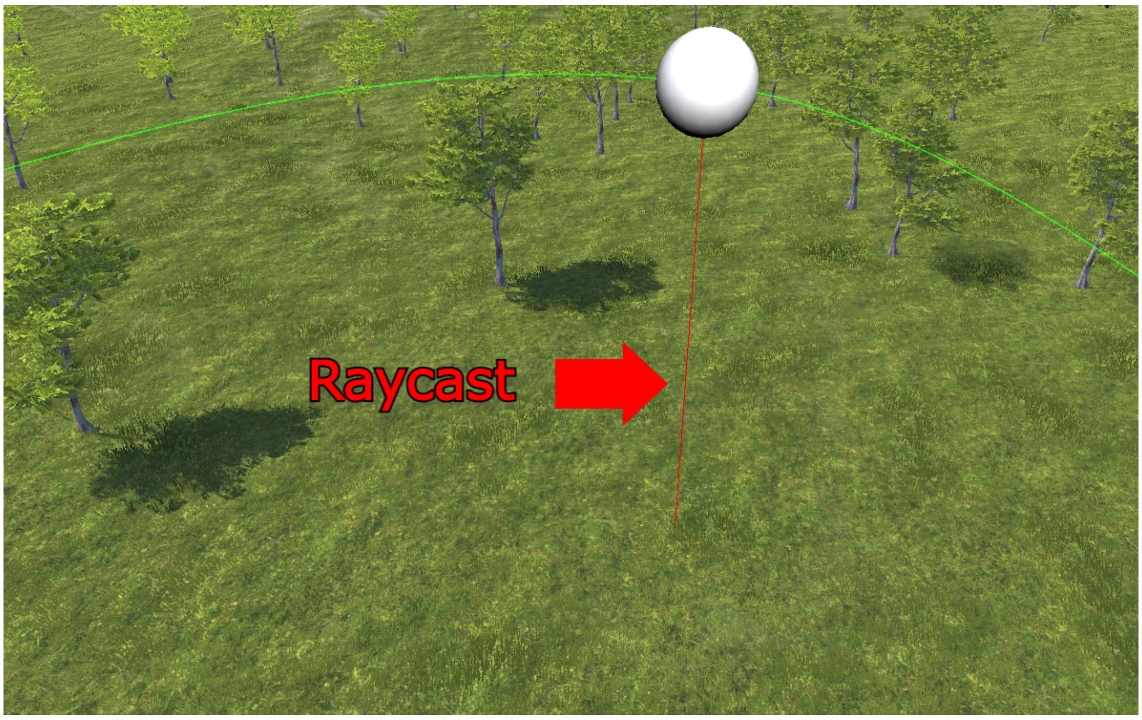


Figure 23: Visualization of raycast (Red) from point of terrain to path line (Green) in Unity Editor

To achieve this we utilized Unity's built-in raycast system. Inside the PathHandle script, we added the code below as an additional event case for when the mouse button is pressed and simultaneously moves:

```
From PathHandle.cs
case EventType.MouseDrag:

    ...

    // Handle can move freely in 3d space
    if (space == PathSpace.xyz)
    {

        Ray mouseRay = HandleUtility.GUIPointToWorldRay(
            Event.current.mousePosition);

        RaycastHit hit;

        Vector3 worldMouse;

        if (Physics.Raycast(mouseRay.origin, mouseRay.direction,
            out hit, raycastDepth, LayerMask.GetMask("Ground")))
        {
            worldMouse = hit.point;
            worldMouse.y += height;
        }
        else worldMouse = new Vector3(0,0,0);

        position = Handles.matrix.inverse.MultiplyPoint(worldMouse);

        Debug.DrawRay(hit.point, Vector3.up * height, Color.blue, .005f);

    }

    ...

    }
    break;
```

This is one of the many modifications coded in the Bézier Path Creator asset and is important because this allows to easily point the mouse on the terrain and set the position of control points for the created path.

6.3.2 Procedural Mesh Generation

After we have defined the path, it should be ready to function together with other objects. However, there was still one thing missing; rendering the appearance of the rails on the path. we first tried to use a method where we place objects that resemble parts of the railings in a set interval between points on the path.

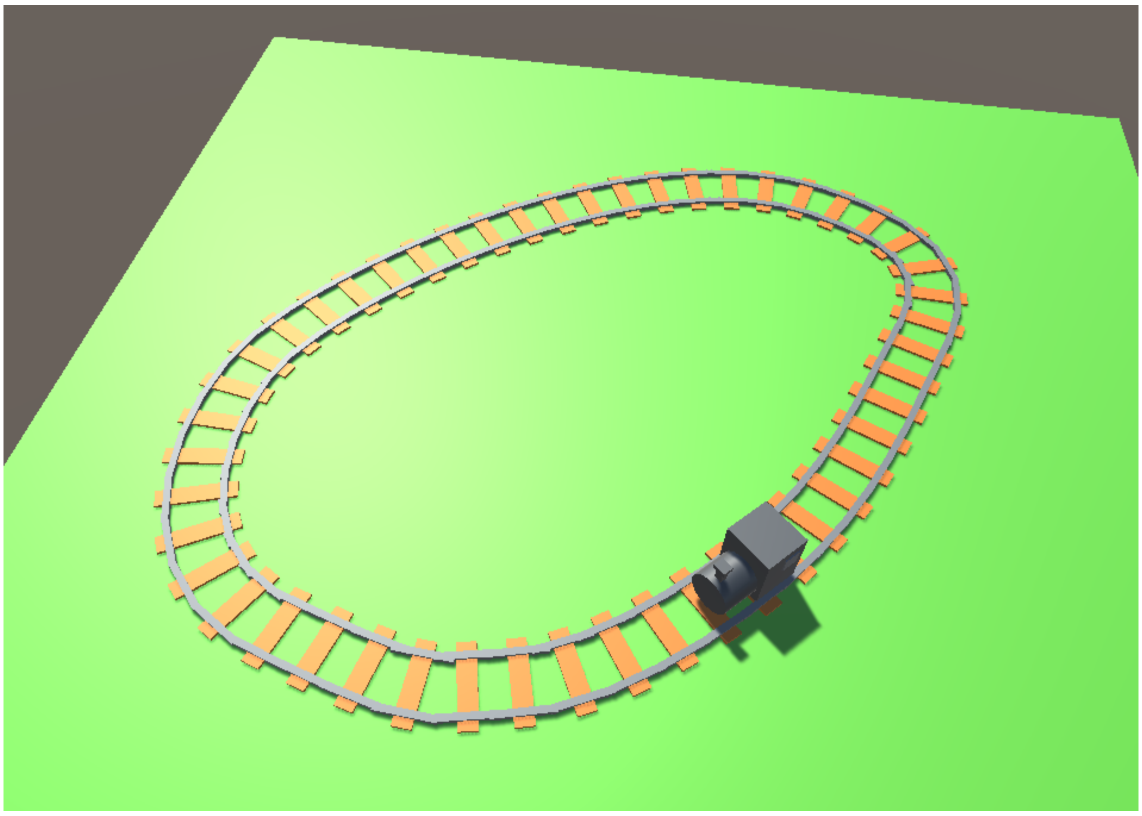


Figure 24: Using prefabs in a set interval to render railings

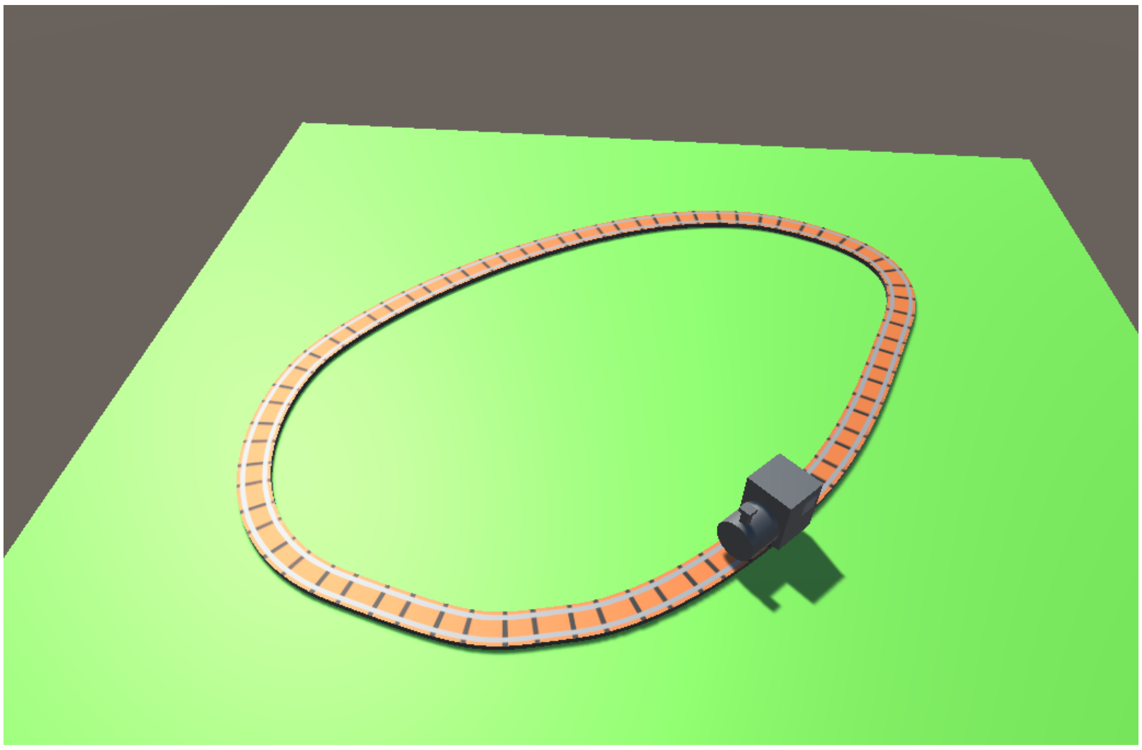


Figure 25: Using procedural mesh generation to render railings

This method presented in figure 24 had a good aesthetic and worked for a short time before we realized that it had some underlying issues. The longer the path was, the more it increased the number of objects in the scene. This quickly led to bad performance and the need for a new solution. In figure 25, the other method we used and kept using from then on comes from the RailMeshCreator script that allows procedural mesh generation on the path. It has set parameters that modify the rail's width, height, and texture, and it worked well without compensating at the expense of performance.

6.4 Input

When it came to choices of getting input we had two options, the old hardcoded way, and the new Unity Input System. The old system was more familiar to us and is far simpler, but it had many downsides, such as it being hardcoded so changes would be very difficult, and its lack of modern VR support. The new Unity Input System provides a lot of benefits that would suit our project, like modularity and complexity of ability, but it also requires a more complex implementation. The new input system unlike the old one supports modern VR through the XR Interaction Toolkit and OpenXR Plugin. It supports all USB HID controllers the user can plug in without more tinkering required [14]. Because of the variety of inputs possible, it is recommended that users implement a rebinding functionality when using this system, and during our testing, we came across the need for this very early, as we discovered that the Saitek Joysticks that the train school uses the XYZ axes for the three sticks on the controller. Windows reads this as emulating the X and Y position of an analog stick, and the Z position of a trigger. Thankfully the ability to rebind is already partially implemented in the Input System but needed some additional work for our use case.

There are a few caveats of the new Input System, most notably that Playstation controllers do not work wirelessly without external drivers and that Xbox One controllers only work on Windows. It also has a few quirks we have had to overcome, notably that there are system-wide input settings that are applied to every input, which are separate from and applied before the general settings for each binding called "processors". These system-wide settings included a 0.125 to 0.95 "deadzone", which means that values below 0.125 or above 0.95 are ignored, which does not work well for joysticks who move from -1 to 1, because this adds a deadzone in the middle of the Saitek sticks. We cannot override this deadzone without removing them completely from every input, which was what we ended up doing, this means that any time a new joystick input is added the deadzone has to be added manually for each binding as a "processor". We also came across difficulties with using two of the same controller. Unity's Input System recognizes controllers based on their name, but when you plug in two of the same controller Windows gives them the exact same name. We had to manually work around this with a toggle in the controller settings, which labels the first joystick you have plugged in as the left-hand joystick, and the second joystick as the right-hand joystick. This is a pretty hard-coded solution, and one that assumes you only have two controllers plugged in, but improving it would require creating a more dynamic system, which we have written more about in section 10 about future work.

6.4.1 Dynamic rebinding

We chose to spend time implementing the ability to rebind all controls because of the great variety of controllers that are commonly used for simulators like this. There are many different simulator controllers and we knew that if the train school decided to use a different controller in the future this would be a good feature to have to guarantee compatibility with any controller, without changing any code. We also prioritized this because after discussions with the train school we knew that they are interested in expanding their leasing of DeskSim to other companies, and when you lease out software like this to other companies the methods of input they have will vary greatly. That is why we implemented the ability to use any kind of controller, from professional sim controllers to regular console controllers, to just a keyboard or just a mouse. Unity's Input System has built-in functions for rebinding, but these are not persistent, so to make the rebindings consistent between exiting and loading the game we implemented our own code around this system. After trying out a few different implementations that did not work for us, we landed on a system greatly inspired by One Wheel Studio's "Adventures In C#" series [11], which is itself a modification of Unity's public sample-implementation of rebinding with the new Input System. What our code essentially does is that it generates the controls from Unity Input System's "Input Action Asset" object (which contains all our action maps, actions, and bindings) like normal, and then we change it during playtime. Every time the InputManager class is loaded it loops over every binding, checks if we have saved an override for that binding in Unity's "player preference" persistent storage system, and then overrides it if so. This system of rebinding works primarily through our InputManager class, which relies on the following functions to create persistent bindings:

- StartRebind - Start the rebinding process for a binding
- DoRebind - Perform a rebinding for an action
- SaveBindingOverride - Save binding overrides to player preferences
- LoadBindingOverride - Load the binding override for action, if it exists
- LoadAllBindingOverrides - Loads binding overrides for all actions that have them
- ResetBinding - Resets a binding

Here are three of the most important functions from InputManager, explained in the comments.

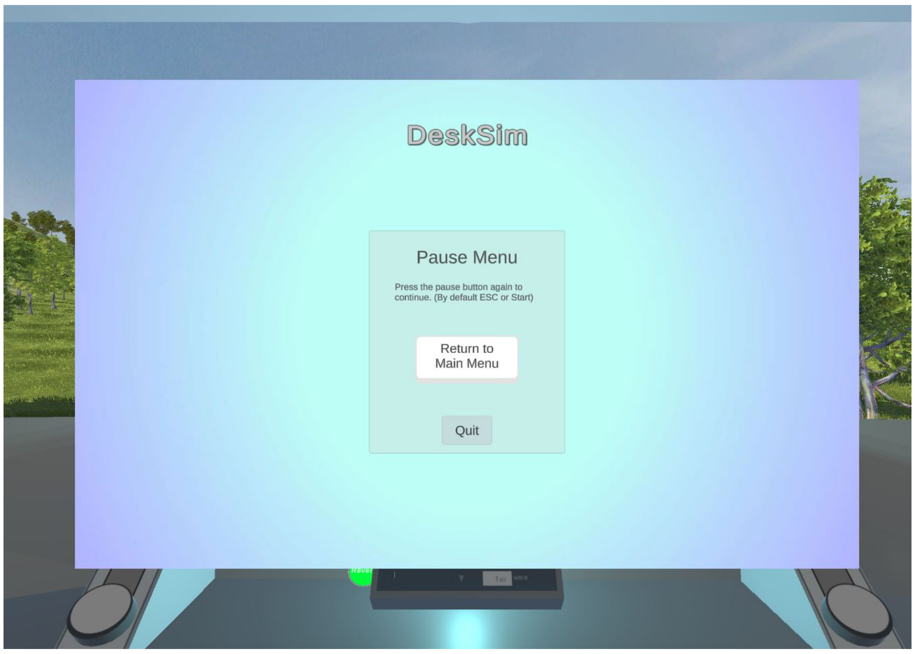
```
    From InputManager.cs
    /**
     * Save binding overrides
     *
     * @param   action  Action to save binding for
     **/
    private static void SaveBindingOverride(InputAction action)
    {
        // Loop through bindings
        for (int i = 0; i < action.bindings.Count; i++)
        {
            // Save binding with key (e.g. "TrainMenu") and binding (e.g.
            ↪  "keyboard/esc")
            PlayerPrefs.SetString(action.actionMap + action.name + i,
            ↪  action.bindings[i].overridePath); // Overridepath is what it
            ↪  looks for during runtime
        }
    }

        /**
      * Load the binding override for an action, if it exists
      *  (Public because it will be called by the RebindUI)
      *
      * @param   actionName  The name of the action to load the binding override
↪   for
      **/
    public static void LoadBindingOverride(string actionName)
    {
        // Initialize userInputActions if it doesn't exist yet
        if (userInputActions == null) userInputActions = new UserInputActions();
        // Load the default bindings for this action
        InputAction action = userInputActions.asset.FindAction(actionName);

        // Loop through all bindings in action
        for (int i=0; i<action.bindings.Count; i++)
        {
            // If a custom binding has been stored for this binding's index
            if (!string.IsNullOrEmpty(PlayerPrefs.GetString(action.actionMap +
            ↪  action.name + i)))
                // Apply the stored binding at current index, using native
                ↪  method
                action.ApplyBindingOverride(i,
                ↪  PlayerPrefs.GetString(action.actionMap + action.name + i));
        }
    }
```

Input System lets us group inputs under ”action maps”, we have an action map for Train controls and another one for Player controls. We have fully implemented the ability to change action maps seamlessly and on the fly in the InputManager script, and for this change to be detectable in other scripts with a C# event. This can be done in the current simulation by pressing ”Enter” by default, this disables the train controls and enables the player controls. Although no camera change is visible, the player's inputs are now being routed to the ”player” action map instead of the ”train” action map, which is picked up by the player controller script instead of the train controller script.

6.5 Menus

The main menu was implemented with standard Unity 2D GUI components, which are a set of components like colored panels, text, buttons, and more, that can be moved and shaped in many ways. The main menu has two different ”screens”, the main menu screen and the scenarios screen, both of which are grouped. We switch between these two by disabling every component of one screen by disabling its group and doing the opposite for the other screen. The controls button is not implemented in the same way as it was created before the main menu, it is a completely separate scene that is loaded when the player presses that button. We kept it this way because to rebind controls the input scripts have to be imported, which is not necessary for the main menu. The scenarios are loaded in the same way, they are separate scenes that Unity switches to when the button to switch are pressed when Unity runs a simple line of code from our Menus file to load a new scene.

There is also a separate in-game menu (Figure 26), which operates as a separate prefab that is simply loaded into every scene we have. Our input scripts that run in every scene are listening for the menu button to be pressed, and when it is pressed, it looks for the pause menu prefab game object in the scene, and if it finds one, shows it to the player and makes it intractable, and pauses the passage of time with a built-in Unity function. Here the player can either exit back to the main menu, by loading the MainMenu scene or continue the game.



Press the pause button again to continue. (By default ESC or Start)

Figure 26: The pause menu

The user interfaces that are used to control the trains are made using the same 2D GUI components. The first GUI we implemented was the one visible when in third-person mode, which is an overlay with two sliders and two gauges, made in the same way as the menus except without a background (Figure 27).

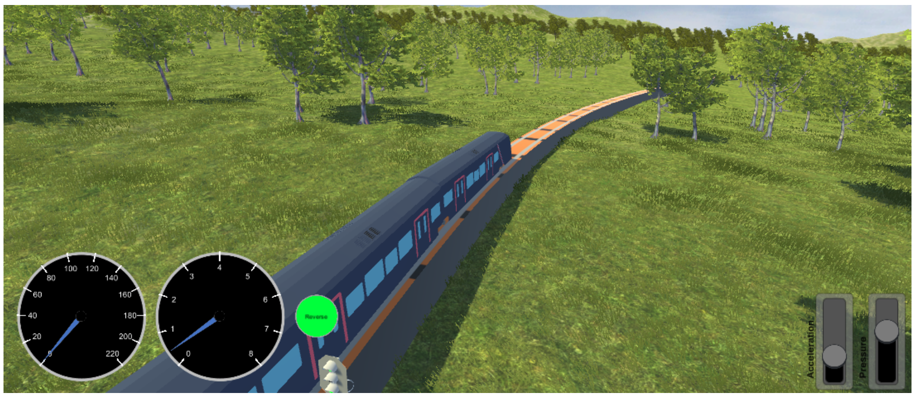


Figure 27: The overlay GUI

The "VR" GUI that is visible in first-person is made with the same components, except these are rendered in world-space and are set to be visible from the first-person camera (Figure 28).

Additional 3D objects like cubes and cylinders are used to create a better 3D effect.

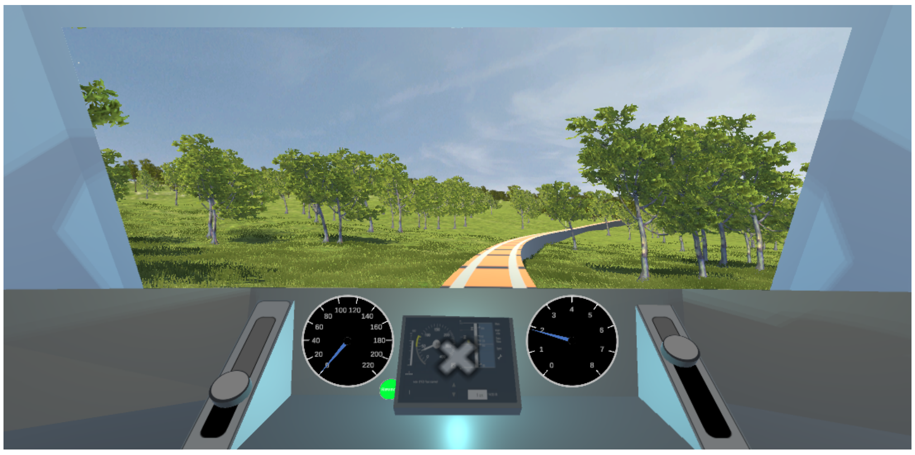


Figure 28: The 3D world-space interface

These two interfaces are synchronized through the TrainInput script, which has access to the sliders from both of these interfaces and updates them both when pressure or acceleration is changed in any way. The gauges are updated through the TrainUI script, which automatically calculates the position for the labels depending on the max value given to them, and moves the labels based on the current pressure and acceleration.

7 Deployment

7.1 Setting up Unity

To download and set up Unity, head over to Unity's official website and download Unity. When configuring the download it will be possible to add extensions and modules, however, we will not go over these as they are purely dependent on the developer's preferences. In our case, we only downloaded the extensions for the development environment Visual studio 2019 and Code. The download should come with the Unity Hub, which you can open after the download, to organize projects and assign unity versions. Here, head to the installs and add the install for the project that we used, Unity 2020.3.28f1. After that, you can also add modules like Visual Studio support and other extensions if necessary.

7.2 Setting up the Project

To open the project in the editor, first, pull the project down from the git repository, and save it on the computer. The repository link can be found in appendix F. Then open the Unity Hub and click the project tab. There you will find the add button and locate the project folder in your file explorer. Before opening it you should make sure that the Unity version for the project is set to the one that you downloaded earlier, Unity 2020.3.28f. If this is not set, it can potentially cause errors to occur because of the difference in the Unity version.

7.3 Creating a scenario

The process of creating a scenario involves creating a new Unity scene based on templates we have created. We considered the option of creating a dedicated scenario editor at the beginning of the project but decided against it as it would be very time-consuming, and would either way not rival the ease-of-use and depth of functionality that Unity's default editor provides. Thus we decided to instead document and streamline the process of creating a scenario with Unity's built-in editor by providing templates, in a way that requires very little knowledge from the user, but which still allows for a great depth of functionality and customizability. Start by opening the project in the file explorer (right-click a folder-¿open in file explorer), go to the "Scenes/Scenarios" folder and copy the scene named "[Template] Scenario" giving it a new name, go into the "MapMagic" folder and do the same for the "[Template] MM Graph" graph file inside there. Now go back to Unity. If you want to base the terrain on a heightmap you need to drag your grayscale heightmap from File Explorer into Unity and drop it in the "Materials/Heightmap" folder, where Unity will automatically recognize it as a heightmap texture. You can source your heightmaps from anywhere, e.g. sites like Tangram's heightmapper. Next, go back to the Scenarios/MapMagic folder and right-click the background→create→MapMagic→Imported map. Select the created object, set "Map Source" to "Texture", and select the heightmap texture you just imported. Now you can go into the MapMagic2 template graph you copied earlier, select the MapMagic heightmap in the "Import" box, and drag from its exit node to the entry node, as illustrated in the picture below (Figure 29).

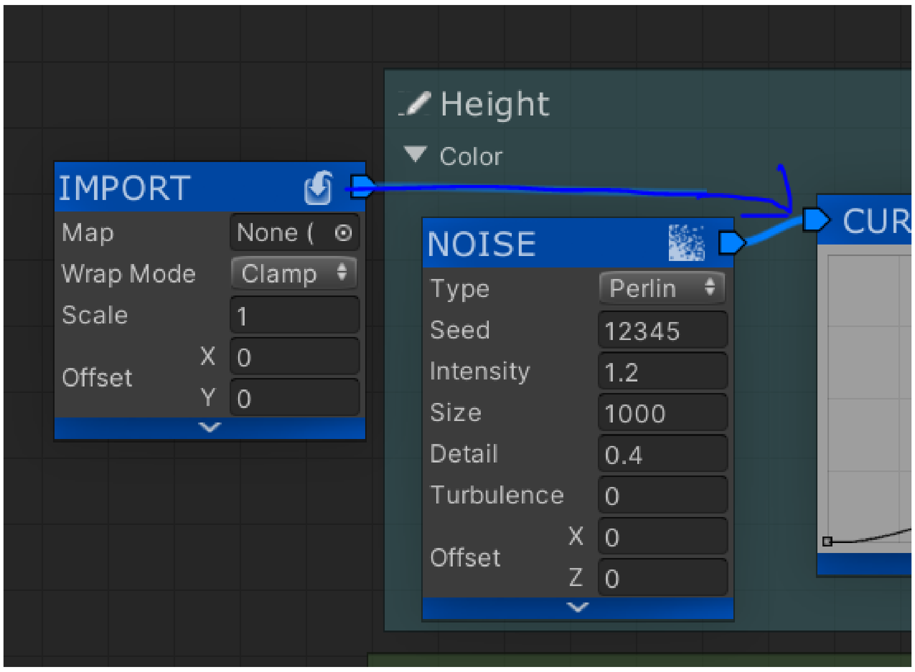


Figure 29: Changing the base from noise to heightmap

Explaining the process of shaping and creating a map with the MapMagic2 graph system is a bit beyond the scope of this project, so instead, I will direct you to the YouTube channel of the developer of MapMagic2, "Denis Pahunov". He has created a playlist called "MapMagic2 Tutorials" which is the best and easiest resource to learn about creating maps with his Unity asset. [12]. To see the tutorial look at appendix F.

Once your scenario is ready to be included in your build, enable the scenario by putting it in build settings, then click File, then build settings and click "add currently open scene" while the scene is open, to add it. To put the scenario in the main menu, go to /Scene/Menu/Main, disable the main GUI by clicking on "Canvas-GUI/Screen-MainMenu" and checking the box in the top right corner. To enable the list of scenarios so you can see it, do the same for "Canvas-GUI/Screen-LoadScenario" (Figure 30).

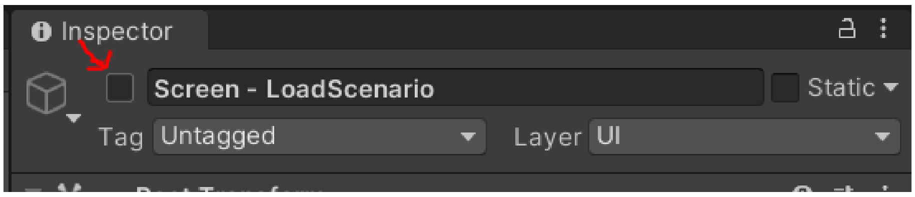


Figure 30: How to enable an element in Unity

Now expand the list of elements down along this path "CanvasUI/Screen-LoadScenario/Panel/ ScrollView/Viewport/Content/List". This is where the entries in the "Load Scenarios" are placed. Copy and paste the object name "ListEntry - Template", expand it and there are three elements, the title, the description, and the button. Click on the title and description, and you can change their contents in the Text component, in the inspector, on the right side. When this is done, click on the button, scroll down in the inspector and change the name in the OnClick() function to the name of the scene you want the button to load. Once this is done, you will want to position the new entry underneath the lowest one currently there. Do this by clicking on the lowest "ListEntry -" object, and noting the value under "Pos Y" in the inspector. Now click on the object you created "ListEntry - Template", look in the inspector and make the "Pos Y" value the previous value you noted minus 115, so it is 115 pixels below the previous box. If you copy the lowest ListEntry instead of the template it will automatically be placed in the same spot as the lowest entry, so then you only have to append "-115" in the "Pos Y" box. If you run the scene and notice that you cannot scroll down to the lowest entry, you will need to increase the "Height" property in the inspector of the "Content" element, "CanvasUI/Screen-LoadScenario/Panel/Scrollview/Content".

Now to finish off, remember to disable the "LoadScenario" screen and enable the "MainMenu" screen, the reverse of what you did in the beginning. Now you can create the build by clicking "File", then "Build Settings".

We have created a video demonstrating this entire process, creating a scenario and putting it in the build, which can be found in appendix F

7.4 Running the build

Start by locating your own build files or our provided pre-compiled build files available on the Release page of our GitHub repository, a direct link to which is available in the appendix F. This contains everything needed to run and test our program. The only thing you need to do is open the "DeskSim.exe" file and you should be greeted with the main menu screen. Before you can start driving the train, you have to select one of our pre-constructed scenarios that have mostly different environments, features, and track layouts. To select a scenario, press the "Load Scenario" button, and then select one of the four playable scenarios to play. After doing this you are put into the train cockpit, from where you can view and test our features. Only the four first scenarios are playable, as the ones below them are non-playable templates scenarios that describe and demonstrate the process of adding more scenarios to the list, they are intentionally cut off.

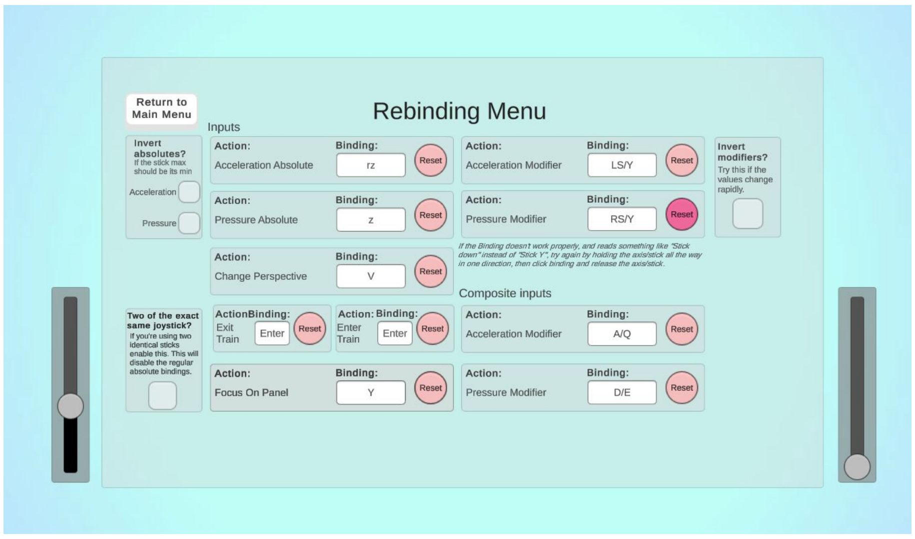
7.4.1 Key Bindings

If you select the "Controls" option in the main menu, you have a full view of the controls with their bindings. This menu is called the rebinding menu and you can select and edit bindings at your own discretion. You can even view in real-time if inputs related to the train control levers are being read by the program by looking at the levers on each side, and they should work with all devices mentioned in section 2.1. We have already set default bindings to what we think works well, but any user can rebind the controls to whatever suits them, with any controller that Unity recognizes. Here is a full table of the default bindings:

Default Key Bindings	
Input	Controls
Acceleration Absolute	rz
Acceleration Modifier	LS/Y
Pressure Absolute	z
Pressure Modifier	RS/Y
Change Perspective	V
Exit Train	Enter
Enter Train	Enter
Focus On Panel	Y
Composite Inputs	
Acceleration Absolute	A/Q
Acceleration Modifier	D/E

Table 1: The default keybindings for the demo

There are also some check-boxes on the rebinding screen, which are used to accommodate for whatever type of input you might have.



Rebinding Menu

Inputs

Invert absolutes?
If the stick max should be its min
Acceleration
Pressure

Action:	Binding:
Acceleration Absolute	rz
Pressure Absolute	z
Change Perspective	V
Exit Train	Enter
Enter Train	Enter
Focus On Panel	Y
Acceleration Modifier	LS/Y
Pressure Modifier	RS/Y

If the Binding doesn't work properly, and reads something like "Stick down" instead of "Stick Y", try again by holding the axis/stick all the way in one direction, then click binding and release the axis/stick.

Invert modifiers?
Try this if the values change rapidly.

Composite inputs

Action:	Binding:
Acceleration Modifier	A/Q
Pressure Modifier	D/E

Two of the exact same joystick?
If you're using two identical sticks enable this. This will disable the regular absolute bindings.

Figure 31: The rebinding screen

The "Invert absolutes" inverts the input of a joystick that has an absolute input, like a lever. So if you have configured a lever to control pressure, but when you push the lever upwards the pressure goes down instead of up, then you should invert the absolute pressure input. The "Invert modifiers" option behaves somewhat differently. We found during our testing that some of our joysticks behaved oddly, in that the joystick value was 0 at the center, but when you started moving it in one direction it instantly read either "1" or "-1" just off the center, and then when the stick was moved to the max and minimum it moved towards 0, which could be illustrated as 0 $\leftarrow$-1(0)1 $\rightarrow$0. This appears in-game as the lever moving slower the further you push your joystick, if your joystick has this quirky behavior as well, enable this, otherwise you should ignore it, and leave it off by default. Lastly, the "Two of the exact same joystick" option is meant to be used for users who are using two of the same joystick, as Unity cannot differentiate between them by default so we have to do some extra work. Currently, this implementation works by assigning the first controller in a list of all connected controllers as the left hand, and the second controller as the right hand. This might break if you have more devices connected that are registered as USB HID Controllers, and you might have to physically swap your right and left controller because the order they are listed by Windows cannot be changed, This solution can be improved in the future, as written about in section 11, Future Work. If you are using two of the same controllers, as two Saitek sticks like the ones the train school uses, enable this option and they will be automatically bound to pressure and acceleration.

8 Testing and User Feedback

We chose not to conduct user feedback, but rather settle for feedback from the train school representative. The reason for this is that the task the train school gave us was to create a program that served simply as a proof of concept. We were tasked with creating a demo that shows that it is possible to recreate the DeskSim simulation program in our chosen game engine, as well as show the possibilities for implementing new features they are interested in using in the future. Because of the specific purpose of this task, we decided that generic user feedback would not serve much of a purpose. We chose not to conduct user feedback and instead prioritized communication with and feedback from the train school representative

Some of the feedback points that we got from the first test that we conducted with the train school representative, on the 23 of march:

- The *skybox* that we used looked weird
- The terrain was lacking in detail, make the terrain prettier.

These concerns were addressed and corrected by finding a more fitting skybox, and by finding a different way to generate the terrain that let us texture it, which is why we switched to MapMagic2. We also added vegetation.

The feedback points that we got from the second and last test with the train school representative, on the 27 of April:

Pros
The simulation feels responsive and fast.
The terrain and environment are nice.
Good looking 3D skybox.
Input works fine.
Cons
The rails looks "Cartoony", but works.
Some problems when trying to map the Saitek controllers where they would cross meaning the acceleration lever would control the pressure and the other way around.

Table 2: The feedback points

The way that the rails look is because of the rail asset that we chose to use for the project, and was touched upon in section 6.3.2 and 9.1.1. The problems related to the Saitek controls seem to be caused by a misunderstanding of the control rebinding screen and our current implementation of Saitek controller support. The current solution to this issue, where the levers are swapped, is to physically swap the Saitek controllers around by removing them from the table and placing them in the correct orientation. The need for this workaround is caused by our current implementation of support for identical joysticks, and as we have already written about how that system should be improved upon in ways that would fix this issue. We decided that we should not prioritize fixing this as it would require too much time. It was a bit unclear exactly what the train school representative experienced was a problem with the controls, so another reason for this mistake could potentially simply be a user error in the controller rebinding screen. We have solved this by making a more in-depth guide on how to properly set up the controllers and bind them in the deployment section 7.4.

One thing we overlooked was user-testing the editor, which was due to the complexity of the test as well as lack of time. This type of test would take a long time to conduct, as the user had to be familiar with the Unity program, to give good feedback on the scenario editing process.

9 Discussion

9.1 Third party libraries

9.1.1 Bézier Path Creator

During the project planning, we spent a lot of time thinking about how to implement the rail system, as it seemed like the most advanced part of the project. We decided to use an asset from Unity's Asset Store for this, and then alter its code to fit our needs. For the purposes of this program being a proof-of-concept, this worked, as it let us prioritize working on other parts of the simulation that show more of the functionality that Unity as an engine provides.

Since the beginning of our development process, we have been using and modifying the free Bézier Path Creator asset for developing the rail system. The asset helped us make the rails very modifiable and easy to edit for anyone, but it also had some problematic issues that in the later stages became more apparent. These issues were solvable but became very time-consuming and unfortunately, we did not solve all of them in the end. Thankfully these issues are not severe enough to warrant a reconstruction of this feature, especially since we made this resolute decision beforehand. We had a time constraint to take into consideration, and changing this underlying code would in effect drastically affect the functionality of the other scripts that relied on the Bézier paths and the architectural structure. However, as said before there are remaining issues, which could even be correlated with the Unity Engine because at one point we changed unity versions in the middle of our development. This happened because we upgraded our Unity version, and pushed it to the main branch, and although updates to Unity are made to be as stable as possible, it is still possible for updates to cause problems.

Although most of the issues with the asset were subliminal and unimportant, it mostly affects the convenience of editing the rails. An example of this is the debug line drawing in editor mode, which draws a line along the path of the Bézier curve, which for some reason causes lag and performance issues when it is rendered. Please refer to figure 20 to visually observe the debug line(in green) that was a causal issue for bad performance. Luckily we found a way around this by disabling the rendering of this debug line by pressing the "hide" button beside the GameObject holding the path and instead used our own rail mesh, which did not cause performance issues. Before coming to this workaround we tried to fix this bug in the code to no avail, and also tried disabling the GameObject whenever it was no longer selected. This solution temporarily worked but caused issues in scripts that were dependent on the path data, which was no longer accessible when the path GameObject was disabled. A bug worth mentioning we found at the last stage of our development, which we regret to say has not been repaired is the rail mesh generation and rendering. Once the rendered rail mesh reaches a certain length it breaks and tries to loop back around to the origin point as if it did not find the right index point it needs to properly follow.

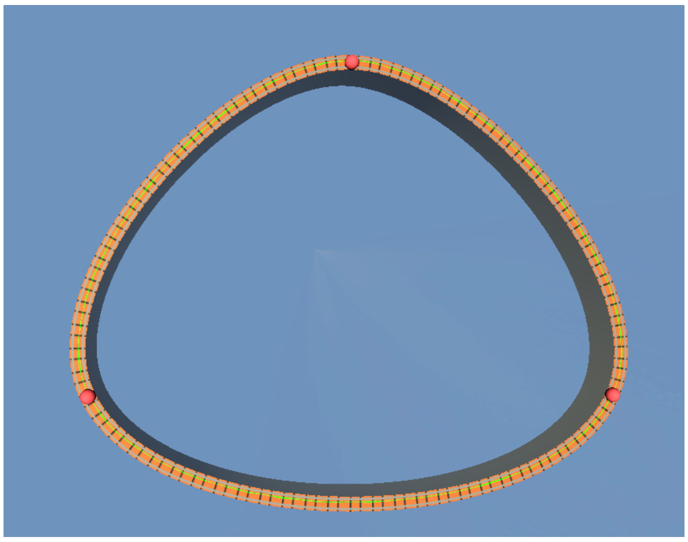


Figure 32: Rail mesh working normally

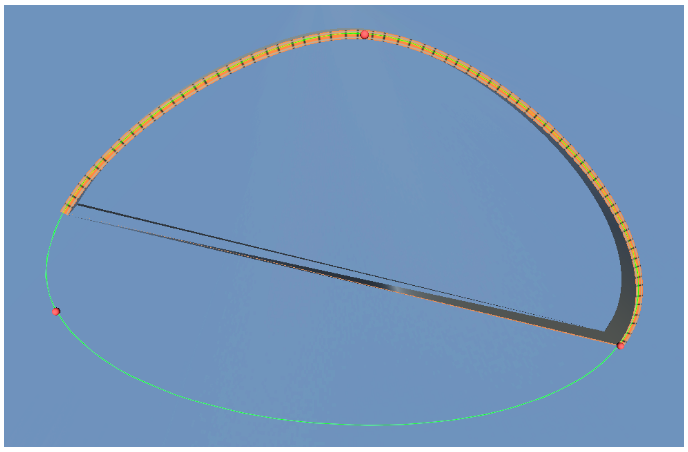


Figure 33: Rail mesh breaking after exceeding a certain distance

The cause of this is unknown, but we had theories that it might occur due to the already mention Unity version change in the middle of development. It still worked, but the version we used was not the official version this asset supports (Unity 2018.1.9 or higher). Testing this phenomenon on a blank project without any modifications to the asset consistently resulted in the same way, it seems to be tied to primarily the distance of the path but is also affected by the distance between each point. The developers have a GitHub repository where users can post issues, and we could not find anyone else with the same issue as us. We believe the unity version change is most likely the cause of this issue since the script for procedural rail mesh generation was logically sound yet persistently had this bug. Our options are to open an issue on this GitHub repository and hope the developers can help us, or we could continue trying to fix it ourselves with more time, or we could switch to a different way of rendering these meshes. Other than this rendering issue everything else regarding our rails functions well, and the issue is unnoticeable if the length of the path does not exceed a certain point, so it is advised to keep this in mind if one wants to change the path for a scenario.

9.1.2 MapMagic2

We decided to use a third-party library to generate the terrain after we found Unity's default tools lacking and requiring too much expertise to use, e.g. needing to write custom shaders to generate textures on terrain. At first, we tried another third-party library for terrain, but also found it unfit for our purposes. The benefits of using MapMagic2 are how easy it is to use once you understand the graph system, and how easily you can change and create new types of terrain and make it suitable for trains, procedurally and infinitely. Another great benefit we did not know about before using it was its ability to import heightmaps, which spared us the pain of finding a separate library for this purpose, or of writing complex shaders to render texture on the terrain. The downsides are few, this is a free yet complex and easy-to-learn and use toolkit with the standard unity license, as the developer monetizes this toolkit with purchasable extensions that expand its functionality. One of the downsides is if we want to use a third-party library to smooth out paths in the terrain automatically because MapMagic generates the many tiles each with a separate Unity Terrain object. This causes problems with third-party libraries in general, because they mostly expect the terrain to be one cohesive terrain object, and will not work well when the terrain is split up into many different terrain objects. So we depend on the developer's implementations and extensions if we wish to use more libraries to modify the terrain.

9.2 Coding and Design

9.2.1 Rails

To extract the information we needed from the Bézier Path Creator asset scripts, we started by creating public "get methods" for the private variables we needed, which lets us get these values without making them public. This would later become tedious, however, as we would have to write get methods for many variables. Therefore we decided to simply switch to using the public method for most of the variables. This was not optimal, as declaring something as the public type can lead to variables being exposed and making the code messy. When making the functionality where the objects snap onto the rails in the editor, this was the case. Therefore, we feel like this part of the rail system is one of the weakest points in the code. To fix this we would modify how the editor script works together with the Follower class, and clean up the code there to use fewer public variables. The classes from the Bézier Path Creator asset and the way they were structured in the hierarchy made them easy to modify for our uses, as they were separated into different subdirectories like "Utility" and "Editor".

The creators of the Bézier Path Creator asset structured it well and made it easy to understand where everything was. The asset separated the scripts and folders into the categories; main functions, editor, utilities, and calculation functions running in the background. For instance, the script calculating the bézier curve algorithm is located in the "Utility" folder. By default, the subdirectories with scripts that are bound and used by Bézier Path Creator were all placed under one directory called "Path Core". This structure was good and made it easy for us to find where the different scripts belonged, but due to referencing problems we encountered in Unity, they are now unorganized. In the last part of our development, we encountered an issue where we could not build and run our program, even though it was functioning correctly when playing it inside the editor. We discovered that this was because scripts that were calling functions from other scripts were not referenced correctly according to Unity. This was a severe problem that gave us a hard time, but after some restructuring, we could finally build and run our program again, at the cost of making our structure slightly messier.

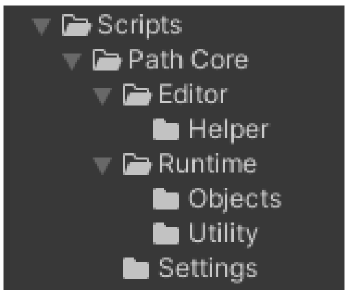


Figure 34: File structure before changes

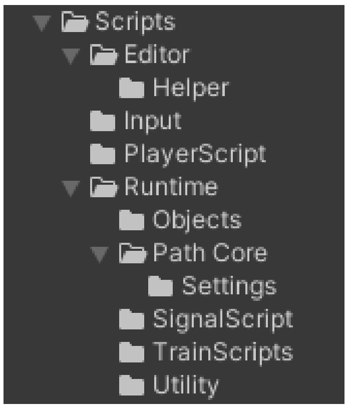


Figure 35: File structure after changes

The "Path Core" folder is supposed to hold all of its scripts and folders like "Editor" and "Helper". These folders are currently outside of the "Path Core" folder. You cannot see the scripts inside the folders from the figures above, but they have changed their location as well. Usually, it is better to practice having third-party library folders isolated, but because of references to one script, "Train Controller", this led to errors related to references. Eventually, we had to fix this severe issue, and figure 35 is what we came up with. Therefore, if there were future development with this project as the foundation, we strongly consider redesigning the structure and as well as the rail system. Though this third-party library was a good solution for creating, managing, and editing rails; it led to more unexpected problems in the end. We had to do several modifications mentioned in section 6.3. The asset has a Github [10] that was from our current time still unfinished and still was being updated. It was optimal enough for our project at the beginning, but it might have been better to re-organize the structure to what we have recommended.

9.2.2 Signals

One of the strong points in the coding is the way the signals are designed for expandability. Using the signals already created as examples and using inheritance, one can easily implement more signals in the future. On the other hand, the SignalController class would be impractical in this program, as we could simply implement it to switch light patterns in its own script. The reason for this is that the script serves as a proof of concept of how the signals can interact with each other through the SignalController class. In the demo, this is shown as the connection between the forward signal and the main signal, as the forward signal's light pattern mimics the main signal's light pattern. Therefore if other signals are connected to each other or affect each other's behavior it can be implemented here.

9.2.3 Train

The train scripts are straightforward and do not have that much in terms of coding complexity, which makes them robust. As the TrainController class is one of the most important scripts and is responsible for the movement of the train, it was developed with some hardcoded values, like the pressure increase and decrease. This means that the script might be vulnerable and open for improvements, as there might be errors that go unseen.

9.2.4 Input

The code that handles input has been written to be very expandable and modular, letting us easily create new action maps and classes which can all coordinate easily with InputManager. The main class, InputManager is however a bit messy in terms of what it is responsible for and could benefit from something like being split into two separate classes, delegating the rebinding functionality to a new class that is instead simply called by InputManager. The TrainInput class also has a bit of legacy code that can be removed.

9.2.5 Error handling

For error handling unity has a lot of options. The one that we used the most during the development phase was the `Debug.Log()` method for debugging and testing purposes. This worked, for the most part, however, we could have probably used more informative methods, to better explain the errors, such as the try-catch method or custom errors.

9.3 Project workflow and communication within the team

As a group, we have worked together on multiple occasions over the course of the bachelor's program, and in general, it has worked fine for the most part. During the project development, we worked individually with our own parts and only came together one or two times a week to reorganize and communicate. This was due to us not being able to have physical meetings, as well as our timetables rarely lined up. To accommodate for individual work, we made a document that contained rules for the work that was to be done over a week. We learned from previous projects, that although we have worked together before and are friends, we should be strict when it comes to project work and keep it professional. In this project, we have managed to do this to a further extent than previously, but one part of the project that would be lacking was the development process.

During the development process is where we feel that we struggled the most. One of the things that we feel we are lacking was enough communication, meaning that we could have benefited from more meetings to catch up. But also we could have used more meetings dedicated to working together online. We think this was crucial, as working together made people more productive as well as making us more motivated. This can be seen in our time log (appendix E), as a spike in hours, as we normally worked together every Monday during the development process. Other than that we normally worked individually but we would also hold more work meetings around our own set deadlines. If we could have had more meetings, we think the project would have improved in some areas, especially when it comes to code design. Looking back we might have benefited from more meetings even if it was for shorter periods of time.

Another thing we could have done better was to be more strict with each other, which also is related to a lack of communication. One of the pieces of evidence is that the time log does not follow the rule for working 25 hours a week, and can be viewed in the time log (appendix E). Some of the lost time is due to the fact that we could not work for that long in the planning phase, as the planning work would not take that long to complete, and we were still lacking information at the start. That way we could only do work in preparation for the actual planning like setting rules and establishing the scrum structure and such. Other cases of lost time were sickness, forgetting to log the work hours, and other things taking priority, as we were also taking other courses during the project. However, the likely reason has to be that we were probably not strict enough with each other. In multiple cases in the time log, group members have not been able to hit the 25-hour mark. The way to fix this would be to more strictly monitor the work of the individual members as well as more communication between the members or have a better reminder of tracking working hours into Clockify. It was something we had never done in our previous projects before and was the main reason we had these problems. Maybe we could have tried to find another solution to this problem, but all things considered, it does not change the fact we all put in our time and a great deal of effort to deliver a good product, and are proud of what we have accomplished.

9.4 Planning

We started our project by organizing a Google Drive folder to contain our project files. From the previous project that we did, we learned a lot when it comes to group management and organizing, and we wanted to put this knowledge to use.

One of the first documents that we made was the ”Regler og Rutiner” document, to see this document look at appendix B, section B.1. This document contained the rules and routines that we were to follow during the project development, and also contact information for all the group members so that they could be reached. During the project development, this document was to serve as a guideline for our group to work effectively and to reflect our work method. Because we were working from home and we wanted to have flexible work times as our schedules never lined up, we had to have rules to ensure that work was being done.

Another initially essential document was the ”Uke Oversikt” spreadsheet. This document showed all of the group member's plans and lectures over the course of a week and served as a reference for the rest of the group when a group member was available. This document did not help us a lot over the project duration, as we would just use the Discord chat to ask questions or organize and plan new meetings in the current meeting. Another flaw is that we made this in the initial stages of our semester and did not have into consideration the hours that were unavailable due to other upcoming projects from other studies group members might have been assigned. However, the document was there if we needed it or if one wondered why we could not get in contact with one another.

The planning phase is one of the most important parts of the project, and it is crucial that it is done right [4]. By making these documents, we made a good foundation for the work to be done. The rules worked less like rules and more of weekly goals to achieve, and it also made communication better by setting at least one set time a week where we would work. That meant that there would at least be one time a week where we could communicate, and talk about the project. Doing this should allow us to work independently while at the same time having a good work ethic and good communication among the members of the team.

10 Conclusion

The goal that we were given by the train school, was to make a demonstrative program implementing the core functionality in their current simulator, in another modern game engine. The demo serves as a proof of concept, proving that DeskSim can be recreated in another game engine, as well as providing a foundation for potential further development. As reviewed in the discussion section (section 9), there are both positives and negatives to our current program, however, most if not all of them are addressed in the report. This is to pass on the information and help future developers to build upon the program, as this is not a finished project. We are pleased that we have achieved most of our goals, and are happy that the project became what we consider practical and convenient. Finally, we are also pleased to hear good feedback from our client, and that they were overall very satisfied with our work.

The development process for this project gave us more insight into design and implementation in general. The project also let us gain more experience with game engines like Unity with its development tools and principles. One concern outside the project itself was to stay consistent with our meeting schedule and log every group member's working hours, which we for the most part accomplished.

Because of lack of time, we, unfortunately, did not develop additional features like rail forks, and a deeper and more robust rail system. But we are still very happy with what we have been able to accomplish up to this point. It has challenged our skills in programming, designing, and cooperation throughout this project and it gave us more hands-on experience. We have experienced that Unity is a great development tool for game developers, but with some limitations. The project will be viable for further development since we laid a good foundation for the train school, should they decide to develop it further. Developing this simulation showed us how solving real-world problems and programming them, never diminished our sense of delight in watching our results come to life.

11 Future Work

This section is an overview of the future and potential developments that we have researched, accommodated for, and taken into account.

11.1 Unfinished work

Though rebinding is fully implemented; a workaround had to be made, to allow for using two joysticks registered by Windows with the same name, as Unity cannot by default distinguish between two of the same controller. After researching the possibility of doing this dynamically, we decided that it was not worth the time spent and hardcoded it, which is why there is a toggle in the settings to enable the use of two identical control sticks at the same time. We would have liked to make this less hardcoded by adding dynamic rebinding for using two identical joysticks so that any two identical joysticks can be used. The way we recommend implementing this is by scanning all connected joysticks and automatically setting identically-named controllers as left and right. More properly this could be implemented by having a button for configuring left and right joysticks, which when pressed prompts the user to move the left stick, then the script listens for changes in the array of all joysticks provided by Unity, and when a joystick is moved it is set as the left controller by giving it an alias, and then this process is repeated for the right stick.

11.2 Future development and research

11.2.1 Character controller

We have fully implemented the ability to change action maps seamlessly and on the fly in InputManager, and for this change to be detectable in other scripts with a C# event. This can be done in the current simulation by pressing ”Enter” by default, this disables the train controls and enables the player controls. Although no camera change is visible, the player’s inputs are now being routed to the ”player” action map instead of the ”train” action map, which would be picked up by the player controller script instead of the train controller script. A full character controller using the player controls is partially implemented and can be enabled when changing action maps, having code to let a player move across varied terrain, but it is not currently in use in any scenarios because of a movement bug and the lack of time to fix it.

11.2.2 Train panel

The creation of a *European Train Control System* (ETCS) screen should be done in the same way as the pause menu, by using a regular Unity prefab with GUI components which can then be duplicated in every train model. This should be a 2D Unity menu that has a script that gets values from the TrainController script and transforms GUI elements in playtime to reflect these values. We researched how to implement a constant measure of the steepness of the tracks early on in the project, which is when we researched the ETCS signaling system used in European trains (Figure 36). This system had been mentioned by the train school when asked about how they calculated elevation, and DeskSim currently uses a similar system.

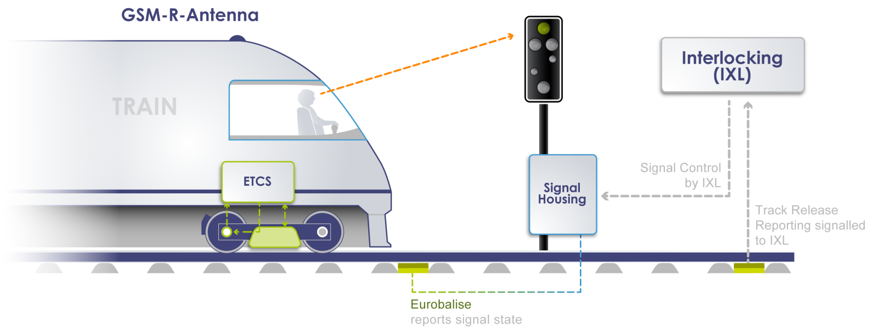


Figure 36: ETCS Signal system [13]

When thinking about how to implement it, the most accurate real-life solution we have thought of is by imitating the ETCS signal system using Unity objects and collision detection. This would be done by creating a prefab object with a box collider, a three-dimensional cube that is used to detect collisions with other objects, that can be spawned at fixed intervals along the Bézier rail path. This could be placed down automatically using the same "DistanceTraveled" value that decides the placement of the train at any given point in time, and then we would automatically or manually generate metadata for each of these boxes by doing things like calculating the elevation difference between each box. The train would then have a separate box collider underneath it, that detects whenever the train is over one of these boxes by checking for collisions, and then it reads the data from them, and displays the information on-screen, just like the ETCS signal system works in real life. If this proved to be too performance inefficient, another way to do this when taking into account how the physics of our train has been implemented would be to simply store these as objects in an array instead of game objects in the world. Then we would give each of these array objects incremental values representing how far along the train tracks they would be placed, which equates to the train's "DistanceTraveled" value, and then use the difference between the two to determine when the train has passed one of these virtual signals. This would mean Unity would not have to do any collision detection because we just use simple comparisons of distances to check when the train would be above a signal. If the entire process of creating metadata for these points along the tracks is completely automated, then there would be no need for storing anything, however, and the code that would generate one of these points would instead just be run at a certain distance interval, like every 50 meters, generating data for a certain distance ahead to show on the ETCS screen. In real life, the values in these signal boxes are manually measured or adjusted, so a fully automated system might not be as flexible as it needs to be, which has to be taken into account.

DeskSim has the functionality of summoning the ETCS screen in a separate window that can be moved around to a different monitor. We are not sure if it is possible to implement this in Unity, but this can be implemented in another way by making Unity span over multiple monitors, and putting the train panel on the second monitor [15]. Currently, we solve this issue by having a button that lets you zoom in on the currently fake ETCS screen, by default with the "Y" button, which toggles a camera that is zoomed in on the ETCS screen.

11.2.3 VR

We have specifically accommodated the future development in VR during our development, and have done a few things to demonstrate that it is possible. The new Unity Input System is a requirement for new VR developments, as it is required for core VR libraries. The Unity Input System is what have implemented, and it is modular enough to implement VR without causing a mess in the code. We have also implemented a pseudo-"VR" GUI, which is a user interface with 3D train gauges and sliders that occupy the world-space inside the cab, which can be made interactable with VR. If the implementation of VR is pursued, then this GUI should be improved to be more appropriate for VR, as the current one is made simply to illustrate that a VR interface is possible, while being designed for flat-screen interactivity. To interact with VR buttons and levers we can write our code or use free or paid Unity Assets designed to make VR interaction more easily implemented. If we were to do it ourselves this is how we would program the levers in VR:

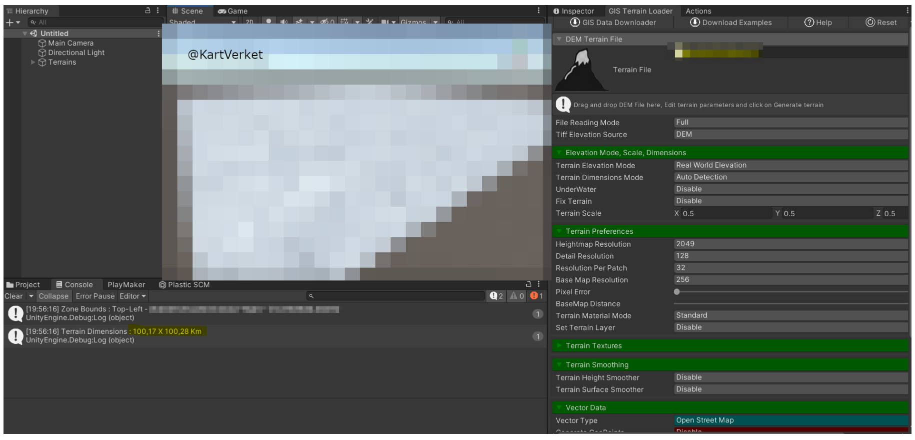
1. Collision checks the player's hand with box colliders on the end of the lever.
2. When colliding, start listening for the trigger button to be pulled
3. If pulled check other conditions if relevant, make the object movable, and call a function to change the value of TrainInput, just like sliders currently do.

Once this kind of interaction system is implemented, adding VR interactions anywhere is easy, such as for the coupling between trains and other things shown in the train school's demonstration of DeskSim. The same is more advanced VR GUI could also be used for flat-screen players for a more realistic interface, like DeskSim does, by simply raycasting under the mouse and seeing if it hits any VR interaction box colliders and if conditions such as distance to the player are met.

11.2.4 Terrain from Kartverket

One of the things we looked into early on was using high detail terrain from Kartverket, as the train school expressed some interest in this. After contacting them about the licensing, we have confirmed that the licensing is the same for our use case, and that to use geographical data from them, a simple credit of some kind has to be on-screen and in the main credits whenever their data is being used. This could be etched somewhere on the desk, it could hover in one of the corners, and in VR it can be on the monitor but out of the VR view. Further inquiry can clarify how we should do this, as ours is currently an unusual use case, and their requirements are vague in what constitutes credit. We also learned from our discussion with them that most of their data have their license, but that there is an exception. When you zoom in to a certain level to a certain fidelity, Kartverket pulls data from what it calls "local projects" like laser scans, in which case the licensing right belongs to the individual municipality or project manager. These local projects are in a list when exporting the data, and you can select or de-select projects, so at the level of fidelity where these projects are included, we would need to contact more people about the licensing. This type of data is often of incredibly high fidelity, so if accurate recreations of real-life locations are desired this might be of interest.

Kartverket offers different kinds of data in many different formats, such as different types of heightmaps, different geographical file formats such as PostGIS, or pointclouds generated from local laser-scanning projects, several of which can be combined into a single pointcloud for download. Reading this data is a complex matter. There are entire courses on it, but it is very valuable as it can render all kinds of data about the terrain, such as rivers and roads, and buildings. So it is a potential avenue for reading geographical data if this kind of data was useful. After looking around we found a way we could use this data without having to create our code for it, using the paid Asset "GIS Terrain Loader" by "GISTech". This asset allows for importing various geographical information systems or GIS formatted files into Unity as plain terrain. We contacted the developer over Discord and talked about whether they would support any of the file types offered by Kartverrket, namely PostGIS, GeoTIFF, USGS DEM, in formats such as DOM, DTM, LAZ, ZLAS, ENH, NED, XYZ. Although we do not know what any of these combinations of letters mean, he was very helpful, and told us that specifically PostGIS was not supported, but that almost all GeoTiff files were supported, and specified that it supported "*.Tiff (16-32-64 Bit) + Tilled Tiff + Multi-Bands + GrayScale". We are not experts on any of this, so to be sure we asked if he could render a GeoTiff sample from Kartverket to see if it worked. He obliged and loaded a 1:1 scale 100x100km piece of terrain for us. (Figure 37)



```
[19:56:16] Zone Bounds : Top-Left - [illegible]
UnityEngine.Debug:Log (object)
[19:56:16] Terrain Dimensions : 100,17 X 100,28 Km
UnityEngine.Debug:Log (object)
```

DEM Terrain File

Drag and drop DEM File here, Edit terrain parameters and click on Generate terrain

File Reading Mode	Full
Tiff Elevation Source	DEM
Elevation Mode, Scale, Dimensions	
Terrain Elevation Mode	Real World Elevation
Terrain Dimensions Mode	Auto Detection
UnderWater	Disable
Fix Terrain	Disable
Terrain Scale	X 0.5 Y 0.5 Z 0.5
Terrain Preferences	
Heightmap Resolution	2049
Detail Resolution	128
Resolution Per Patch	32
Base Map Resolution	256
Pixel Error	
BaseMap Distance	
Terrain Material Mode	Standard
Set Terrain Layer	Disable
Terrain Textures	
Terrain Smoothing	
Terrain Height Smoother	Disable
Terrain Surface Smoother	Disable
Vector Data	
Vector Type	Open Street Map

Figure 37: Picture of GeoTiff data from Kartverket imported as terrain with GIS Loader. Blurred to avoid potential licensing issue, as we don't need to show the data, only to demonstrate that importing it is possible.

The provided picture shows GeoTiff data we exported from Kartverket and sent to the developer of GISTech, loaded as the terrain in Unity. Although the licensing shouldn't be an issue as long as @KartVerket is shown on screen, we cannot remember which zoom level this data was extracted from, so all data has been blurred just to be sure. This simply because it's not important to see the data for the purposes of this report, this is only to demonstrate that it can be imported. Inquiring about the coloring of the terrain he said "GTL is able to generate texture based on ColorRamp shader. Or you can set your custom texture (Sat, top ... etc)" meaning that there are many options for coloring the terrain based on geographical data or elevation if some time is spent researching it.

Currently, we can download most kinds of heightmaps from the internet, import them into Unity with MapMagic2, scale them to approximately 1:1 by manually doing measurements, and then texture them procedurally, as detailed in the Deployment section 7. For this extension to be worth the price, it will need to be able to offer more than what we can currently achieve with this free method, like rendering roads and buildings or higher fidelity heightmaps. We think this extension can do this, but it would be wise to further inquire with the developer to confirm this functionality before purchasing the extension, as it is somewhat uncertain.

The creator of the GIS Loader asset also has a couple of other Unity Assets, one of interest being the GIS Streamer asset, an asset that allows for procedural streaming of GIS data. This means that you can select any location on earth, and the asset will automatically download and render the GIS data for that region. The licensing of the streamed GIS data is uncertain, it claims to use open sources and might have a way to select which source to stream from. If the licensing is figured out, and usable for the train school, this would in practice let us generate terrain from the earth anywhere, place rails and objects down on this terrain, and theoretically continue this around the entire earth. Our rail system is compatible with this, as we store the rail as a simple Unity GameObject with scripts attached. The positions of rail and all objects should be consistent when the terrain is cleared off and loaded again. This slightly depends on if data is pulled from different sources for the same area and if those sources have different measurements, in other words, it depends on the data being consistent. If it is, this means we should be able to place down tracks anywhere, go to another place entirely and place rails there, and then go back to the first place where we placed the rails. At a scale like this, some modifications should be done to the rail system to acquire enhanced performance at long distances, like culling the mesh and implementing a fix for a bug with the rendering of the mesh. A downside of this asset is that there is no way to manually change the terrain as it is just going to be cleared off and reloaded. A solution is to overlay pieces of Unity terrain separate from the real terrain to do so, which is doable, and like the other Unity objects, will stay in the same place no matter what geographical area you load.

11.2.5 MapMagic2 modules

Denis Pahunov, the creator of the terrain generator we used, MapMagic2, has released many different assets that work as additional modules that can be bought to expand the functionality of MapMagic2. This is the business model that allows him to offer the base module for free. "MapMagic2 Objects" allows the procedural placement of objects such as primarily vegetation, but it can likely be used for buildings as well. We would recommend switching over to MapMagic2 objects if budget allows, as the asset we currently use to spawn trees, "Vegetation Spawner", is not very well integrated with MapMagic2. The "MapMagic 2 Brush" asset gives access to a lot of brush tools to manually change more of the terrain in ways that are persistent even after changing the terrain generation by changing the noise graph. The method we currently use to manually change the terrain is by using Unity's built-in tools, but those changes are overridden by MapMagic2 when re-generating the terrain, so we cannot change the procedural shape of the terrain without resetting the manual changes made to it. The asset "MapMagic2 Splines" is perhaps of the greatest interest, it allows for the automatic creation of long "splines", letting you draw long lines in any shape over any kind of bumpy terrain, and then it creates a smooth path underneath those lines. The splines module would allow for automatically creating smooth paths for rails to go over, though this module is still in beta. There are many other spline Assets in the Asset Store, but according to reviews, they have issues with MapMagic2 as they cannot make splines that span over multiple MapMagic2 tiles, because each tile is considered a separate piece of terrain by Unity, so it is best to stick to the first party Splines module.

Bibliography

1. Flafla2. Newtons lover. 2014. Available from: https://adrianb.io/2014/08/09/perlinnoise.html. (Found 2.5.2022)
2. Observation ER and Center S(. USGS EROS Archive - Data Use and Citation. 2018. Available from: https://www.usgs.gov/centers/eros/science/usgs-eros-archive-data-use-and-citation?qt-science_center_objects=0#qt-science_center_objects. (Found 19.5.2022)
3. Grøn Ø. Newtons lover. 2021. Available from: https://snl.no/Newtons_lover. (Found 2.5.2022)
4. Team IE. What Is Project Development? With Steps and Tips. 2021. Available from: https://www.indeed.com/career-advice/career-development/project-development. (Found 29.04.2022)
5. tog N. Type75. 2021. Available from: https://www.norsketog.no/tog/type75/bmb75. (Found 17.02.2022)
6. Wirtz B. jMonkeyEngine: For Those Who Are Just Getting Started or Who Wants To Build Simpler Games. 2021. Available from: https://www.gamedesigning.org/engines/jmonkey/. (Found 6.04.2022)
7. Clockify. The most popular free time tracker for teams. 2022. Available from: https://clockify.me/. (Found 25.04.2022)
8. Creations S. Unity Asset - Vegetation Spawner. 2022. Available from: https://assetstore.unity.com/packages/tools/terrain/vegetation-spawner-177192. (Found 9.2.2022)
9. Lague S. Bézier Path Creator Asset. 2022. Available from: https://assetstore.unity.com/packages/tools/utilities/b-zier-path-creator-136082#description. (Found 8.05.2022)
10. Lague S. Bézier Path Creator Github. 2022. Available from: https://github.com/SebLague/Path-Creator. (Found 8.05.2022)
11. OneWheelStudio. Key Rebinding With Unity's New Input System and Generated C# Code. 2022. Available from: https://www.youtube.com/watch?v=TD0R5x0yL0Y. (Found 13.2.2022)
12. Pahunov D. YouTube Playlist - MapMagic2 Tutorials. 2022. Available from: https://www.youtube.com/watch?v=-fO8P-dp28E&list=PL8fjbXLqBxvbsJ56kskwA2tWziQx3G05m. (Found 29.1.2022)
13. Thales. European Train Control System (ETCS). 2022. Available from: https://www.thalesgroup.com/en/european-train-control-system-etcs. (Found 29.1.2022)
14. Unity. Input System 1.3.0 Manual - Supported Input Devices. 2022. Available from: https://docs.unity3d.com/Packages/com.unity.inputsystem@1.3/manual/SupportedDevices.html. (Found 23.2.2022)
15. Unity. Unity Manual - Multi Display. 2022. Available from: https://docs.unity3d.com/Manual/MultiDisplay.html. (Found 3.4.2022)

Appendices

A Appendix A

Kontaktperson: Isak Kvalvaag Torgersen

Epost: isator@jernbanedirektoratet.no

Forslag til bacheloroppgave innen IT fra Lokførerskolen

Simulatorutvikling

Norsk fagskole for lokomotivførere, eller Lokførerskolen, er en offentlig statlig godkjent fagskole som utdanner lokomotivførere til hele landet. Som en del av utdanningen bruker vi en selvutviklet programvare kalt DeskSim for simulering av togframføring og andre relaterte oppgaver. Denne programvaren er bygget med Java i spillmotoren jMonkeyEngine og 3D-modelleringsverktøyet Blender brukes for å lage modeller.

Oppgaven

Spillmotoren jMonkeyEngine har eksistert siden 2003 og selv om det fortsatt er i bruk og vedlikeholdes så er ikke engasjementet rundt dette prosjektet det samme som det har vært. Deler av jMonkeyEngine begynner å bli utdatert og Lokførerskolen vil unngå å havne i den situasjonen at verktøyene vi bruker ikke lenger er vedlikeholdt eller støttet. Dagens situasjon med tanke på moderne spillmotorer som er åpent tilgjengelig for bruk er også en helt annen enn når utviklingen av DeskSim ble startet. Vi har derfor startet et langsiktig prosjekt for å kunne overføre DeskSim til en moderne spillmotor.

Oppgaven består av to deler:

1. **Innledende del;** undersøke hvilke alternativer som finnes med tanke på moderne spillmotorer som er offentlig tilgjengelig og gjøre en faglig vurdering av fordeler og ulemper med de forskjellige alternativene.
2. **Hoveddel;** gjenskape elementer av DeskSim ved hjelp av den spillmotoren som gruppen vurderer som det beste alternativet. Denne delen vil være delt inn i et hovedmål og flere delmål. Delmålene kan studentene jobbe med dersom de ser at hovedmålet er oppnådd.

Oppgavens mål

Målet med oppgaven er å gi Lokførerskolen et godt grunnlag for å vite hvilken spillmotor vi skal satse på i framtiden.

Oppgavens krav

Innledende del

- Kartlegge og sammenfatte informasjon om moderne spillmotorer som kan være aktuelle for Lokførerskolen å bruke.
    - Spillmotorene må tilfredsstille følgende krav:
        - Spillmotoren **må** være 'moderne'.
        - Spillmotoren **må** gjøre det mulig å gjenskape de eksisterende elementene i DeskSim.
        - Spillmotoren **må** være egnet for VR-utviling.
        - Spillmotoren **bør** gjøre det mulig å gjenbruke eksisterende ressurser slik som 3D-modeller fra DeskSim.
        - Spillmotoren **bør** være så lett som mulig å lære seg
    - Funnene skal presenteres i en oversiktlig og lettfattelig rapport som inneholder følgende:

Kontaktperson: Isak Kvalvaag Torgersen

Epost: isator@jernbanedirektoratet.no

    - Informasjon om viktige faktorer om spillmotorene slik som programmeringsspråk, lisensmodell, osv.
    - En begrunnet vurdering av hvordan de forskjellige spillmotorene etterlever Lokførerskolens krav og hvilken spillmotor gruppa anser som det beste alternativet.
  - Vi ønsker at gruppa skal se på **minst** tre alternativer inkludert Unreal Engine og Unity.

Hoveddel

Hovedmål

- Lage en demo i den spillmotoren som gruppa anser for å være det beste alternativet. Demoen må tilfredsstille følgende krav:
  - Demoen må inneholde minst et tog, minst to signaler, en tog-DMI som ligner på den i DeskSim, togspor og et enkelt landskap. Det er en bonus hvis man kan gjenbruke ressurser fra DeskSim.
  - Toget må kunne startes og stoppes av bruker ved hjelpe av spakene som Lokførerskolen bruker i dag.
  - Toget må kunne kjøre en kort strekning hvor det følger togskinnene på en realistisk måte.
  - Signalene må kunne endre seg underveis (for eksempel skifte fra rødt til grønt eller vis versa) basert på forhåndsbestemte faktorer (slik som togets posisjon).
  - All koden som brukes for å lage demoen må være veldokumentert og det bør være mulig å bygge videre på den koden.
  - Gruppen bør følge etablerte prinsipper for programvareutvikling/-design og gjøre rede for valgene de tar.
  - Det kan være nødvendig å justere eller tydeliggjøre kravene underveis og det forventes at utviklingen skjer i tett samarbeid med Lokførerskolen
  - Det bør være fokus på at verktøyet bør være så enkelt og intuitivt som mulig å bruke.

Delmål 1

- Lage et verktøy for å plassere ut 3D-modeller (tog, signaler, bygninger, osv.) langs forhåndsdefinerte jernbanestrekninger.
  - Det må også være mulig å flytte eller fjerne eksisterende modeller.
  - Når nye modeller har blitt plassert langs en strekning må de være mulig lagre endringene slik at de blir reflektert når man skal kjøre strekningen neste gang.

Delmål 2

- Lage en strekningsbygger. Med strekning menes en virtuell strekning med jernbanespor hvor det er mulig å kjøre simulerte tog.
  - Det må være mulig å opprette, redigere og fjerne strekninger.
  - Strekningene må kunne inneholde kurver (ikke bare rett fram).
  - Når en strekning er opprettet må det være mulig å plassere ut en togmodell og kjøre strekningen.
  - Det er ikke et krav at det er mulig å endre på landskapet i strekningen eller justere på høydekurver i sporet o.l., men det må tas høyde for at dette skal bli mulig senere.

Video som illustrerer DeskSim finner du her; https://lokforerskolen.net/tmp/bachelor/

B Appendix B

Kunnskap for en bedre verden

Fakultet for informasjonsteknologi og elektroteknikk

PROG2900 - Bacheloroppgave

Prosjekt Plan

Author:
Marco Ip
Julian Kragset
Michael-Angelo Karpowicz

31, januar, 2022

Innholdsfortegnelse

1 Mål og Rammer

1.1 Bakgrunn

Lokførerskolen bruker spillmoteren jMonkeyEngine som har eksistert siden 2003, og selv om den er fortsatt i bruk og vedlikeholdes så holder Lokførerskolen ikke mye engasjementet rundt prosjektet som før. Grunnen er at deler av jMonkeyEngine blir gradvis mer og mer utdatert og stopper å støtte nye teknologier, dvs. at Lokførerskolen begynner å åpne seg til andre alternativer. Målet våres er å designe og bygge en simulasjonsprogram på en moderne spillmotor som Lokførerskolen potensielt kan fortsette å utvikle. Vi iverksetter prosjektet med hensikt til å utrette og fullføre bacheloroppgaven våres og tilby en fin løsning til oppgavegiveren. Vi vil tilby spillmotoren Unity som er moderne og veldig anvendelig i dag. Vi fikk tildelt å konstruere en simulasjonsdemo fra Lokførerskolen og NTNU. Simulering skal mesteparten være ment for å undervise blant studenter og instruktører uten risiko.

1.2 Resultatmål

Det vi vil foreligge når prosjektet er ferdig er å skape en intuitivt, men samtidig nøyaktig simulasjon av å være en lokfører. Lokførerskolen får da teste og bruke en modernisert simulering de kan potensielt bruke i fremtiden på DeskSim. Vi vil vise alle mulige fordeler og egenskaper med Unity sin spillmotor.

1.3 Effektmål

Effektene og gevinstene vi vil gi til lokførerskolen er en trygg, effektiv, og så nærme som mulig realistisk simulering for studenter. Dette fører til en mer lærerikt og grundig utviklingsmiljø. Prosjektet vil også inneholde muligheten å konstruere og plassere ut tog, jernbanestrekninger, signaler, osv. for å skape avgjørende scenario-er. Dette kan øke engasjementet for Lokførerskolen å utvikle videre prosjektet. Over tid kan Lokførerskolen utvikle programmet som en rammeverk for VR-simulasjon, fordi Unity støtter det og forbedrer opplevelsen for studenter.

1.4 Læringsmål

Gjennom prosjektet skal vi oppnå kunnskap og kompetanse innen flere områder, men hovedsakelig skal det i størst mulig grad være prinsipper innen programmering:

- Prosjektplanlegging og gjennomførelse av prosjektet
- Gjennomgang av objektorientert design
- Kodegenerering i C#
- Veldokumentering av koden og avklart bestemmelser innen programvareutvikling/-design

2 Oppgavebeskrivelse

Hoveddelen av simulasjonen skal inneholde minst et tog, to signaler, en tog-DMI (Driver Machine Interface) som skal ligne på den i DeskSim, togspor og et enkelt landskap. Toget skal være operativ og kunne starte og stoppe av brukeren vha. spakene Lokførerskolen bruker. Signalene skal kunne endre seg på forhåndsbestemte faktorer. Det er også delmåler som har mindre prioritet for oss, men som vi er åpen for hvis vi har gjort hoveddelen allerede i en stor grad. Første delmål er å kunne lage et verktøy som skal plassere ut tredimensjonale modeller langs forhåndbestemte jernbanestrekninger. Andre delmål er å lage en verktøy for å plassere jernbanestrekninger på forskjellige måter som for eksempel i kurver og krysser. Vår undersøkelsesmetode for utviklingsprosessen blir å lese den velskrevet dokumentasjon fra Unity, eksterne kilder og eventuelt konsultering med veilederen.

2.1 Omfang og avgrensning

Prosjektet vårt består i hovedsak av to deler. Den første delen handler om å gjøre en analyse av forskjellige spillmotorer og skrive en rapport hvor det sammenlignes minst 3 spillmotorer. Her skal de følges noen fastsatte krav ettersom klienten ønsker å bruke prosjektet til videreutvikling. Spillmotoren skal være morderne, det skal være mulig å gjenskape elementer fra DeskSim i den nye spillmotoren, den skal være egnet til VR-utvikling og man skal kunne bruke en joystick som input for simulatoren. Etter analysen så skal det komme en konklusjon over hva gruppen mener er beste alternativet for spillmotoren. Den andre delen handler om å bruke den valgte spillmotoren til å lage en demo som skal ligne DeskSim, og har krav som vi utviklere skal følge. Vi har også fått oppgitt delmål for om vi blir ferdige, som er handler om å utvikle verktøy relatert til laging av scenario.

Det er følgende områder vi ikke skal prioriteres innenfor programmets rammer:

- Kompetanse til regelverket innen jernbaner
- Høy kvalitet i grafisk gjengivelse
- Virtuell realitet
- Detaljert landskap og terreng

2.2 Fagområde

I simulasjonsutvikling er det behov for flere områder vi må ta i betraktning. Vi må kunne bygge opp og utvikle selve programmet fra bunnen av fordi vi bestemte oss å utvikle simulasjonen i Unity. Dermed er hovedkompetansene vi skal kunne er følgende:

- Programmering og simulasjonsdesign i Unity
- Grafikk og visualisering
- Matematikk, fysikk og mekanikk
- Plasserings- og modelleringsdesign innen 3D-områder
- User interface design

3 Prosjektorganisering

3.1 Organisasjonskart

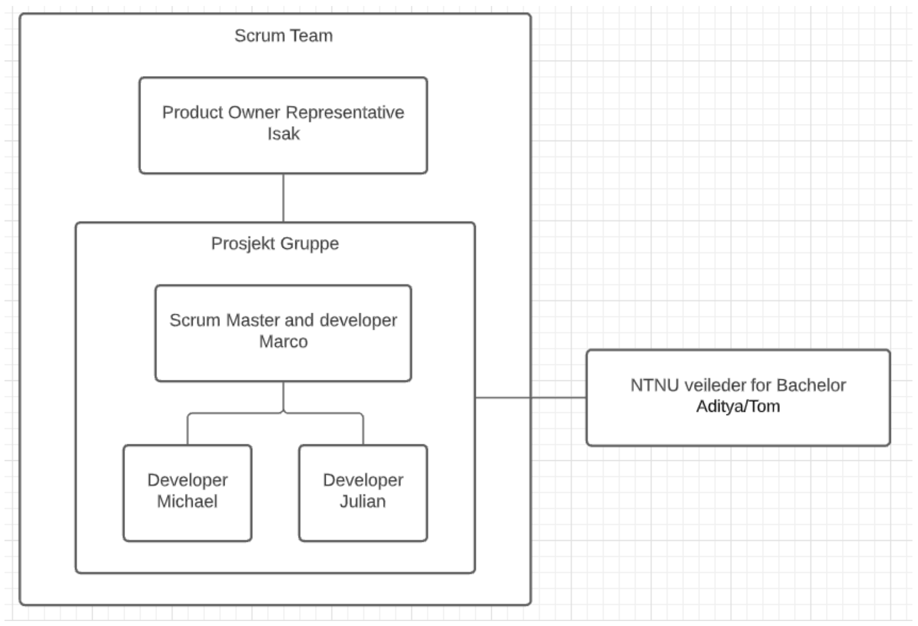

Scrum Team
Product Owner Representative Isak
Prosjekt Gruppe
Scrum Master and developer Marco
Developer Michael
Developer Julian
NTNU veileder for Bachelor Aditya/Tom

Figure 1: Organisering av Scrum team og involverte parter

3.2 Ansvarsforhold og roller

- Scrum Master: Marco Ip

  Gjennom et tidlig møte har blitt enige som en gruppe at Marco overtar rollen som gruppeleder og scrum master for prosjektet. Han har da ansvar for innkalling til møter både med gruppen, veileder og lokførerskolen, samtidig som å organisere og delegere arbeid og holde orden på gruppe dokumentene. Gruppeleder er og ansvarlig for kommunikasjon med partene involvert i prosjektet.

- Developers: Michael-Angelo Karpowicz, Julian Kragset, Marco Ip

  Utviklerne har ansvar over å holde frister og følge regler og rutiner satt i et eget dokument i gruppedokument mappen. Deriblant det er satt regler og rutiner for arbeidstimer, prosjektinnleveringer, møter og egendager. De skal være generalister ettersom vi er en liten gruppe, og gjør at vi kan jobbe mer fleksibelt.

- Product Owner Representative: Isak Kvalvaag Torgersen

  Representanten er ansvarlig for å sette opp møter etter behov og svare på spørsmål som gruppen har til utviklingen av simulatoren. Er og ansvarlig for å ta opp eventuelle forespørsler med lokførerskolen.

- NTNU veileder: Aditya Sole, Tom Røise

  Veileder er ansvarlig for å følge gruppen gjennom prosjektet og eventuelt veilede og gi råd på diverse punkter som innleveringer og rapporter. I begynnelsen så vil Tom stå for veiledning, men senere så vil Aditya overta.

3.3 Rutiner og regler i gruppa

På prosjektstart så har det blitt laget en rekke dokumenter som skal hjelpe gruppen med å klargjøre og effektivisere arbeid. Disse dokumentene ligger i en mappe på Google disk og forteller om regler og rutiner, har oversikt over timeplanene til medlemene og linker til resursser som vi bruker. I dokumentet med regler og rutiner så er det skrevet om arbeidstimer hver uke, rutiner rundt prosjekt innleveringer, møter og organisering av timeplan og egendager. Vi har også et dokument som holder på alle linkene til resursser vi aktivt bruker. Det vil si for eksempel Github repoet, trelloen vår hvor vi holder styr på issue-tracking og produkt backloggen vår og overleaf dokumentene vår, for å nevne noen.

Noen viktige punkter fra regler og rutiner, se vedlegg 1.

3.4 Verktøy

Disse er verktøy vi bruker for å hjelpe oss gjennom prosjektet:

- **Unity Game Engine** - som vår spillutviklings motor.
- **Visual studio/Vs Code** - som IDE.
- **GitHub** - for lagring og deling av prosjektet.
- **Trello** - for å holde styr på sprint prosessen og sprint backloggen.
- **Google Disk** - for å holde styr på gruppe dokumenter.
- **Google Calander** - for å holde gruppen oppdatert om innkommende hendelser og møter.
- **Overleaf** - brukes for å skrive rapporter og innleveringer.
- **Clockify** - for å registrere arbeidstid.
- **Teams** - for møter med veileder og lokførerskolen.
- **Discord** - for interne gruppemøter.

4 Planlegging, Oppfølging og rapportering

4.1 Hovedinndeling av prosjektet og metodikk

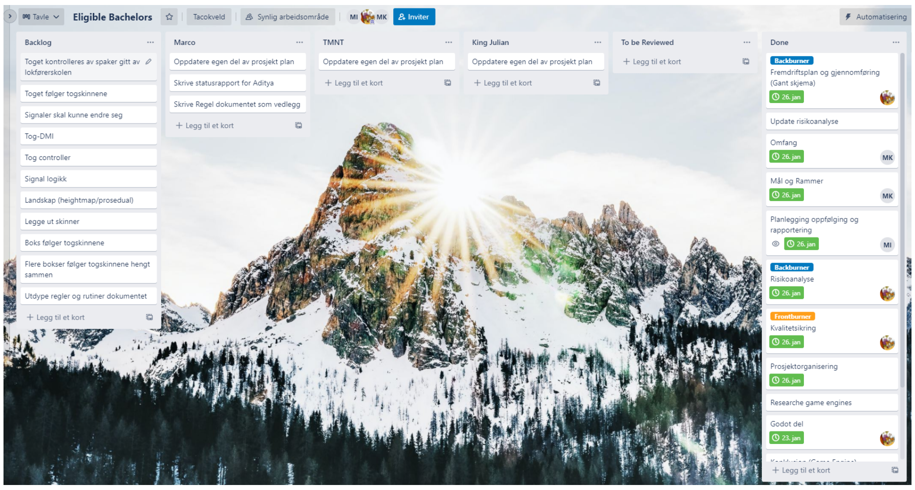
Vi har bestemt oss for å bygge vårt prosjekt på scrum metodikken, ettersom tidligere erfaring med denne type prosjekt har vist at scrum er et bra rammeverk som passer gruppens arbeidsvaner. Prosjektet vårt består i hovedsak av to deler. Den første delen handler om å gjøre en analyse av forskjellige spillmotorer og skrive en rapport hvor de sammenlignes. Den andre delen handler om å bruke den valgte spillmotoren til å lage en demo som skal ligne DeskSim.

Vi er ferdig med første delen av oppgaven og skal nå gå over på andre delen av prosjektet. Scrum vil bli en viktig del av denne fasen ettersom mye arbeid kommer til å bli gjort individuelt. Dette er for å kunne være fleksibel i henhold til gruppens individuelle planer. Vårt mål med scrum er å kunne jobbe med prosjektet individuell samtidig som at det skat være tett oppfølging på utviklingen. For å kunne oppnå dette har vi valgt å holde en ukes sprinter, og minst en felles arbeidsdag i uken.

For nå så er andre del av oppgaven delt inn i tre store deler (se Gannt skjema, Figur 3). Disse er tog kontroller, togskinn system, landskap og annet. Disse skal beskrive de overordnede oppgavene våre gjennom prosjektet, men skal utbredes senere, når vi skal planlegge selve prosessen og lage domene og arkitektursmodell.

4.2 Utførelse av Scrum som metodikk

Ettersom vi skal følge scrum metodikken så må vi lage en produkt backlogg og siden vi må følge kravene til lokførerskolen, så kan vi enkelt gjøre kravene om til produkt backloggen vår, og legge til ekstra oppgaver etter behov. Dette skal da utarbeides under prosess planleggingen i en senere tid. For å holde oversikt over sprintene og sprint backloggen så bruker vi Trello, hvor vi har oppgaver som individuelle kort vi kan flytte rundt på, og gjør sprintene oversiktlige.



Backlog
- Toget kontrolleres av spaker gitt av lokførerskolen
- Toget følger togskinnene
- Signaler skal kunne endre seg
- Tog-DMI
- Tog controller
- Signal logikk
- Landskap (heightmap/prosedual)
- Legge ut skinner
- Boks følger togskinnene
- Flere bokser følger togskinnene hengt sammen
- Utdype regler og rutiner dokumentet

Marco
- Oppdatere egen del av prosjekt plan
- Skrive statusrapport for Aditya
- Skrive Regel dokumentet som vedlegg

TMNT
- Oppdatere egen del av prosjekt plan

King Julian
- Oppdatere egen del av prosjekt plan

To be Reviewed

Done
- Backburner — Fremdriftsplan og gjennomføring (Gant skjema) — 26. jan
- Update risikoanalyse
- Omfang — 26. jan — MK
- Mål og Rammer — 26. jan — MK
- Planlegging oppfølging og rapportering — 26. jan — MI
- Backburner — Risikoanalyse — 26. jan
- Frontburner — Kvalitetsikring — 26. jan
- Prosjektorganisering — 26. jan
- Researche game engines
- Godot del — 23. jan

Figure 2: Organisering av Scrum ved bruk av Trello

Under backlog så ligger alle oppgavene vår som skal gjøres for at prosjektet ska kunne sies ferdig. Dette er produkt backloggen vår så langt hvor vi allerede har lagt inn kravene fra oppgaven, og om flere oppgaver skulle dukke opp så vil de legges inn her. Sprint backlogen er representert som oss tre hvor man kan se hvem som holder på med hvilke oppgaver. Så har vi en review kolonne, hvor oppgaver som er ferdige skal tas opp og sees over på neste sprint møte. Til sist har vi en ferdig kolonne som holder på alle oppgavene son har blitt gjort ferdig. Vi har valgt denne måten å strukturere prosjektet på ettersom det gir oss en god oversikt over oppgavenes status. Vi valgte også å dele sprint backloggen i tre for å kunne enkelt se hvem som jobber med hva og hvor mange oppgaver en har, man skal også markere oppgavene slik at vi kan se hvem som har gjort hva.

Under scrum prosessen så vil hver sprint vare i en uke, som forklart tidligere, hvor vi jobber med oppgavene vi har fordelt på Trello. Etter en uke skal det holdes et møte hvor vi legger fram det vi har fått til, det som ligger i review kolonnen, eventuelt hvor langt vi har kommet på oppgaven. Dette skal gjøres for å få god oversikt over hvor i prosjektet gruppen ligger og hvor fokuset burde ligge. På slutten av møte skal nye oppgaver bli tildelt til de som ikke har oppgaver og Trello-en skal bli oppdatert for å reflektere endringene under møtet.

4.3 Oppfølging

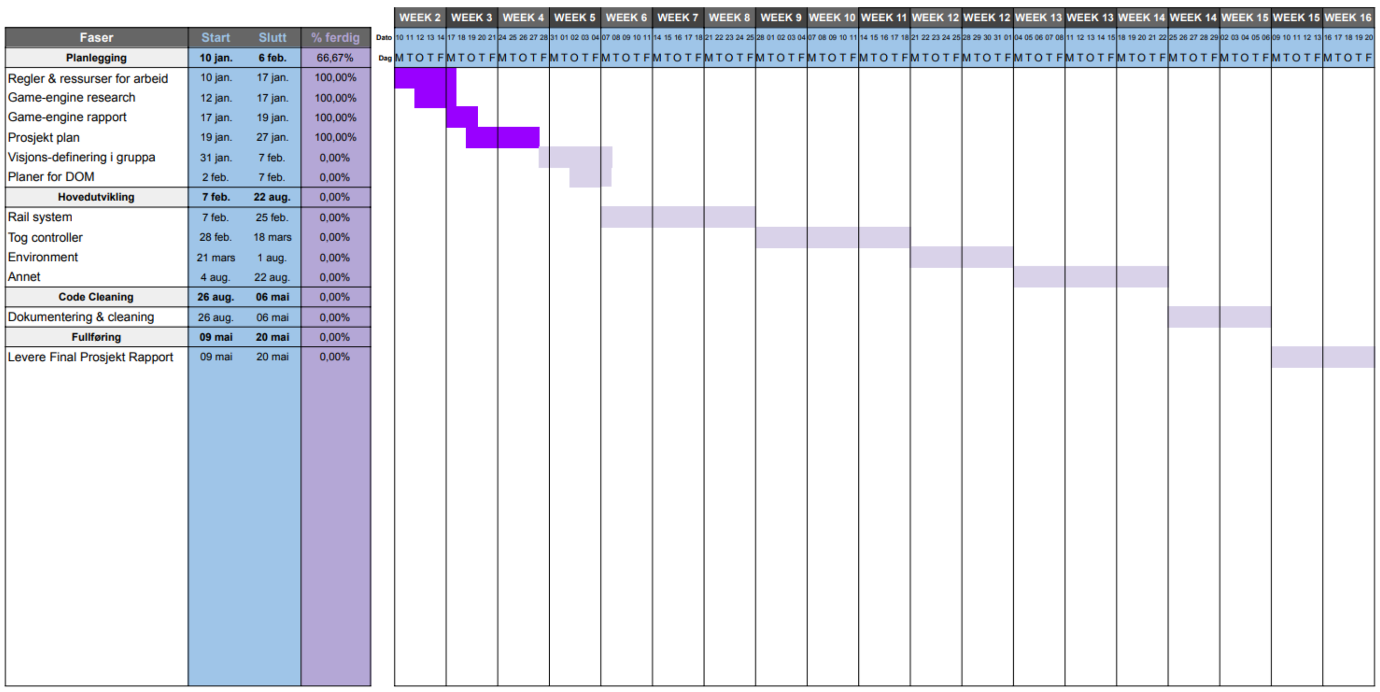
For prosjektet så skal det holdes møter med både veileder og lokfører representant minst en gang i måneden, med flere i begynnelsen og når det nermer seg innlevering. Mengder møter vil dermed variere men for nå, så har vi møte med lokfører representant annen hver mandag kl 13:00 og med veileder hver mandag kl 14:30. Antall møter kan øke eller reduseres etter behov og avtale mellom partene. Under møte med lokførerskolen så skal det først presenteres hva som har blitt gjort etterfulgt av hva som trengs å gjøre, slik at representanten vet hvordan prosjektet ligger an. Etterpå så vil det være mulighet for spørsmål til representanten angående utviklingen. For møte med veileder skal de også presenteres arbeid hittil og eventuelt planer frammover, men hovedfokus vil ligge i utarbeiding av raport og andre saker relatert til bacheloren.

Gruppen har som avtalt å ha minst en fast samlet intern arbeidsdag på mandag hvor vi skal samles og jobbe i et digitalt rom. Denne dagen vil da starte med Scrum møtet hvor vi går over arbeidet.

Om det trengs så kan gruppen selv kalle inn flere arbeidsmøter internt i gruppen, men det er hovedsaklig gruppeleder som styrer disse.

5 Gantt skjema

Gannt skjemaet er ikke en plan vi kommer til å følge slavisk, men mer et estimat for når vi kommer til å jobbe med hva, og en illustrasjon av progressonen. Vi kommer til å jobbe mer fritt i hendhold til scrumm, og kommer til å planlegge underveis som vi jobber, da vi ikke har nok erfaring med spillutvikling til å gi nøyaktige estimater for fasene vi kommer til å gå gjennom. Denne planen er altså et estimat.



Faser	Start	Slutt	% ferdig
Planlegging	**10 jan.**	**6 feb.**	66,67%
Regler & ressurser for arbeid	10 jan.	17 jan.	100,00%
Game-engine research	12 jan.	17 jan.	100,00%
Game-engine rapport	17 jan.	19 jan.	100,00%
Prosjekt plan	19 jan.	27 jan.	100,00%
Visjons-definering i gruppa	31 jan.	7 feb.	0,00%
Planer for DOM	2 feb.	7 feb.	0,00%
Hovedutvikling	**7 feb.**	**22 aug.**	0,00%
Rail system	7 feb.	25 feb.	0,00%
Tog controller	28 feb.	18 mars	0,00%
Environment	21 mars	1 aug.	0,00%
Annet	4 aug.	22 aug.	0,00%
Code Cleaning	**26 aug.**	**06 mai**	0,00%
Dokumentering & cleaning	26 aug.	06 mai	0,00%
Fullføring	**09 mai**	**20 mai**	0,00%
Levere Final Prosjekt Rapport	09 mai	20 mai	0,00%

Figure 3: Gannt skjema

6 Kvalitetssikring

For å kvalitetssikre prosjektet vårt skal vi følge solide rutiner og regler rundt utviklingen, som god Git etikkette og felles kode-gjennomgang.

Når en feature er implementert og gjort ferdig av en eller flere medlemmer så går vi i neste møte over arbeidet som er gjort, og ser om det passer til planen og visjonen vi har for prosjektet. Vi skal følge en robust Git strategi for å forsikre at ingen feil kommer inn i hoved-grenen av Git repositoriet. Vi skal ha en hoved-gren som alltid skal inneholde en fungerende versjon av programmet, en ”development” gren som vi bruker til å utvikle i, og nye grener for hver feature. Dette forsikrer at uansett om feil skjer i utviklings-grenen så vil det ikke påvirke hoved-grenen, som alltid vil ha en fungerende versjon av programmet.

Når vi skal utvikle en ny feature så lager vi en ny gren ut fra utviklings-grenen, som er dedikert til den nye featuren som skal utvikles. Alt arbeid på den nye featuren skjer på den grenen, som betyr at det ikke har noen risiko for å kollidere med andre sin kode mens man utvikler. Når featuren til denne grenen er helt ferdig og i fungerende stand så skal denne grenen ”merges” inn i utviklings-grenen og slettes, gjennom en ”Merge-request” som andre skal se gjennom. Men før denne merge requesten lastes opp til Git repoet skal utvikleren teste den selv, og forsikre seg om at deres nye innspill til utviklings-grenen fungerer som planlagt uten å ødelegge andre features i utviklings-grenen. Hvis andre gjør forandringer på utviklings-grenen mens man utvikler på sin egen ”feature”-gren så har man selv ansvar for å merge inn forandringene som er gjort i utviklings-grenen inn i sin egen ”feature”-gren og forsikre seg om at det også fungerer. Når dette er bekreftet så kan utvikleren åpne en ”merge-request”, hvor de andre medlemmene skal gå inn og se over forandringene som er gjort i koden, og teste grenen selv, før de kommenterer på eventuelle feil som må ordnes før mergen kan aksepteres, eller bekrefter mergen. Når alt er løst kan merge-request’en aksepteres, og da er den nye featuren en del av utviklingsgrenen vår.

Når vi er enige om at vi har gjort nok framdrift i utviklings-grenen i et av framdrift-møtene våres, så oppdaterer vi hoved-grenen vår. Da merger vi inn forandringene vi har gjort i utviklings-grenen inn i hoved-grenen. Dette vil på lik måte bli testet av merger-ansvarlig før de merger, og alle andre medlemmer gjennom merge-requesten, for å passe på at det fungerer på alle sine systemer og at ingen feilaktig versjon av programmet kommer inn i hoved-grenen. Hvis programmet ikke kjører riktig for et av medlemmene så vil dette bli fikset samlet i merge-request stadiet, før vi prøver igjen. Hvis av en eller annen grunn en feil kommer gjennom denne prosedyren og disse testene så vil vi bruke Git versioning for å reversere merge'n og finne ut av problemet sammen, og starte en ny merge-request. Vi skal også ha en kultur av hyppige og beskrivende commits, slik at feil kan finnes og reverseres når det blir nødvendig.

7 Risikoanalyse

	Risiko	Sannsynlighet 0-3	Konsekvens 0-3	Tiltak
	Tap av kode	Sannsynlig	Moderat	Hyppig bruk av Git, hyppige commits og opplasting av kode
	Game-breaking bugs	Sannsynlig	Moderat	Solid code-reviewing prosedyre og hyppige Git commits så feilen kan finnes og reverseres
	Prosjektet blir ikke fullført til fristen	Usannsynlig	Kritisk	Scrum med ukentlige progress-checkups
	Gruppemedlem(er) blir syke over lengre tid	Usannsynlig	Kritisk	Forandring av scope og evaluering av prioriteter
	Prosjektet beveger seg for sakte i forhold til planen	Mulig	Seriøst	Forandring av scope og evaluering av prioriteringer
	Tap av rapport som ved utilsiktet sletting av Overleaf filer	Mulig	Seriøst	Ukentlige lokale backups av rapporten
	Lisensen passer ikke lenger programmet	Ubetydelig	Moderat	Flytte prosjektet til nytt lisens

B.1 Vedlegg 1. Regler og Rutiner

Regler og Rutiner

Gruppeleder

Gruppeleder skal stemmes fram mellom gruppemedlemmene og har ansvar for at følgende regler og rutiner blir fulgt. Gruppeleder har og rett til å delegerer arbeid utover gruppen. Gruppen kan også velge ny gruppeleder om flertallet stemmer for det.

Krav om arbeidstimer pr uke

Som Standard så skal det legges minst 25 timer med arbeid for prosjektet pr uke, og dette skal logges med Clockify.

Unntak

Dersom vedkommende er syk eller av andre grunner ikke kan fullføre arbeidsmengden spesifisert, skal det meldes til alle gruppemedlemmene og argumenteres med årsak.

Konsekvens

Dersom årsaken er dømt utilstrekkelig av de andre gruppemedlemmene så er det forventet at arbeidet som gjenstår, skal tas igjen i løpet av den neste uken.

Møter

Hver uke skal det holdes et møte med veileder som er obligatorisk for alle. Kan endres dersom veileder og gruppen mener at det ikke trengs. Ellers så er det også obligatorisk å møte på forhånd avtalte møter med gruppen. Dette gjelder timer satt opp for arbeid sammen online.

Det er også obligatorisk for alle å møte opp på møte med representant for lokfører skolen, hvor det og vil være åpent for endring av antall møter.

Prosjekt innleveringer

Minst en uke før innleveringer skal det avtales et møte hvor vi diskuterer og jobber med innleverings rapporten. Før fristen så skal det sendes inn minst ett forsøk til veileder for å sjekkes, før den endelige rapporten er levert.

Egen dager

Dersom vedkommende ønsker eller vet at de ikke er mulig å få tak i, gjelder vanlige kontakt metoder(discord, messenger etc.), og ønsker ikke å bli forstyrret, så burde man markere perioden på "Uke Oversikt".

C Appendix C

Concept

Action:
An action is an occurrence like an event, however it is always user generated.

Asset:
In a Unity context, an asset is an item that you can use in your project.

Bézier curve:
A Bézier curve is a curve that is represented by using a number of control points as guides to define a curve that has no mathematical representation.

Branch:
A branch is an independent repository split off from the main repository that is to be independently developed.

Component:
In the Unity context is a functional piece of every game objekt, meaning that to give an object a functionality you give it an component.

DeskSim:
Desksim is the train school's currently running game simulation program that helps with teaching new train drivers.

Domain Model:
A domain model is a conceptual model of a domain that displays its functionality with concepts, roles and relationships.

European Train Control System:
The European Train Control System or ETCS is the signaling and control system used on European railways.

Events:
Events in programming are sudden occurrences that can be detected by programs, and can be either system or user generated.

Game Engine:
A game engine is a software that is used to create games or simulations in either 2d or 3d.

Github:
Github is a platform where groups can collaborate and share code with each other amongst other things.

Graphical User Interface:
Graphical User Interface or GUI is a type of user interface where the user can interact with software through graphical icons.

Hardcode:
To hardcode is to fix a program so that it can not be altered without modifying the program.

Heightmap:
A heightmap is an image, often a gray-scale image, that stores surface elevation in every pixel on the image.

Hitbox:
A hitbox is an invisible area where objects coming into contact with the box will trigger something like an event.

InputActionAsset object:
The InputActionAsset object is an asset that contains the action maps as well as other control schemes.

Input Action Map:
Input Action Map is a series of input actions that are collected and treated as a group.

Integrated Development Environments:
Integrated development environments or IDE is a software that uses a GUI to contain code and programming tools for the developer to use.

Inspector:
In Unity, the inspector displays detailed information about the currently selected GameObject as well as public variables, Serializefield variables, and components.

JMonkeyEngine:
JMonkeyEngine is the game engine that DeskSim is built on.

Lokførerskolen/Norsk fagskole for Lokomotivførere:
A vocational school that trains students to become train drivers. For more information visit their website at https://lokforerskolen.no/

Mesh:
A mesh is any geometry in a space that consists of triangles.

Noise:
Noise is an error that is overlayed on top of the true value.

Normal:
In geometry, a normal is an object such as a line that is perpendicular to a given object surface.

Prefab:
A prefab is a saved game object that often has child objects or components on it so that it can be reused.

Product Backlog:
A product backlog is an ordered list where all the tasks that have to be done in order to finish a project are stored.

Raycast:
A Raycast is a projection of a beam that returns a boolean whenever it collides with something.

Repository:
A repository is an online place where files may be stored.

Runtime:
Runtime is the phase when the game is running.

Saitek Controllers:
Saitek Controllers are the controllers that the train school uses. The full name is “Logitech G Saitek Throttle Quadrant”.

Scenario:
A scenario is a prebuilt world where you try to complete the tasks that the world is making the user do.

Scene:
A scene is an asset in Unity where you can work with content, like making a game level or a world.

Scrum:
Scrum is a framework that helps teams to organize and work effectively together.

Shader:
Is a program that computes color and light onto an object when rendering a 3D scene.

Skybox:
An image that covers the sky, giving the illusion of surroundings that aren’t there, like clouds and distant terrain.

Spline:
A spline is a generated curve that moves through multiple points.

Sprint:
One sprint is the duration between each sprint meeting, often from one to two weeks.

Sprint Backlog:
A sprint backlog is an ordered list where all the tasks that are being worked on during a sprint are stored.

Sprint Meeting:
A sprint meeting is a meeting where the tasks in the sprint backlog are taken out if they are done, and new tasks are assigned.

Unity:
Unity is a modern game engine.
https://unity.com/

Unity Editor:
The Unity editor is where you develop and create games using Unity’s tools.

Unity object/GameObject:
An abstract "thing" that can be given different properties, like a position, a script, a hitbox, a model, physics, can be whatever.

Virtual Reality:
Virtual Reality or VR is a simulation technology that allows the user to interact with software in a 3D environment.

D Appendix D

NTNU
Kunnskap for en bedre verden

Fakultet for informasjonsteknologi og elektroteknikk

PROG2900 - Bacheloroppgave

Game Engine Rapport

Author:
Marco Ip
Julian Kragset
Michael-Angelo Karpowicz

17, januar, 2022

Innholdsfortegnelse

1 Introduksjon

Idag så ligger lokførerskolen under jernbanedirektoratet og er med på å utdanne togførere for hele Norge. De gjør dette gjennom bruk av morderne teknologier som simulerings kammer, spill og VR teknologi ved siden av vanlig klasserom undervisning. Lokførerskolen bruker spillmoteren jMonkeyEngine som har eksistert siden 2003, og selv om den er fortsatt i bruk og vedlikeholdes, så ser de etter nye måter å forbedre simuleringen sin. Grunnen er at deler av jMonkeyEngine blir gradvis mer og mer utdatert og stopper å støtte nye teknologier, dvs. at Lokførerskolen begynner å åpne seg til andre alternativer, og det er disse vi skal diskutere her.

I analysen så er visse krav fra lokførerskolen som skal være med for utvalg av spillmotor. Spillmotoren skal være morderne, det skal være mulig å gjenskape elementer fra DeskSim i den nye spillmotoren og den skal være egnet til VR-utvikling. Et annet punkt som vi fikk vite senere er at man skal kunne bruke en joystick som input for simulatoren. Disse kravene er det tatt full hensikt til og hvis en spillmotor ikke oppfølger ett av disse kravene så ville den ikke vært med i analysen. Andre krav som vi fikk tildelt var at spillmotoren burde kunne gjenbruke allerede eksisterende ressurser som objektfiler og 3D modeller fra DeskSim, og burde være rimelig lett og lære. Gjennom analysen så vil vi da legge mest vekt på disse, men også se på andre egenskaper, og velge en motor å bruke til prosjektet.

2 Unity Game Engine

Unity er en 3D/2D spillmotor og en kraftig cross-platform IDE for utviklere. Som en spillmotor er Unity i stand til å tilby mange av de viktigste innebygde funksjonene som får et spill til å fungere. Det betyr ting som fysikk, 3D-gjengivelse, og kollisjonsdeteksjon. Fra en utviklers perspektiv betyr dette at det ikke er behov for å finne opp hjulet på nytt. I stedet for å starte et nytt prosjekt ved å lage en ny fysikkmotor fra bunnen av; beregne hver eneste bevegelse av hvert materiale, eller måten lyset skal sprette av fra forskjellige overflater. Unity bruker C# for å håndtere kode og logikk sammen med en haug av klasser og API-er som er enhet til Unity.

Fordeler:

- **Kan bruke .glTF og .obj filer** Opprinnelig så støtter Unity blant annet .obj filer, men ikke glTF. Dette kan da løses med en tredjeparty plugin som det bare er å hente fra Unity sitt samfunnsmarked.
- **Lett å lære** Unity har som mål å forenkle spillutviklingsprosessen, og editoren taler for det. Dokumentasjonen er praktisk og hjelpsom i alle områder for å bygge en god prosjekt. (Unity 2022a)
- **Virtuell Realitet** Unity er en fantastisk spillmotor for virtuell virkelighet. Unity hevder å være den foretrukne plattformen for å skape VR-opplevelser. Bransjeledende utvikleropplevelse med deres høy høy optimaliserende verktøy for redigering av pipeline og raske iterasjonsmuligheter gir oss alle de avanserte funksjonene vi trenger for å utvikle en bra VR-opplevelse. (Unity 2022b)
- **Bra for å gjengi 2D og 3D-scener** Unity er en av de mest populære moderne spillmotorene for spillutvikling(Perforce 2020). Det er veldig effektiv når den gjengir 2D og 3D-scener. Kvaliteten er relativt bra sammenlignet med andre spillmotorer.
- **Bra fellesskap og god dokumentasjon** Unity sitt fellesskap av erfarne utviklere og dokumentasjon er det beste ressurs vi kan bruke når det gjelder å forstå utvikling. Dokumentasjonen er detaljert og er en stor hjelp for å sette i gang de riktige handlingene under spillutvikling.
- **Lavere kostnad** Til tross for at den ikke er åpen kildekode, er kostnadene lavere sammenlignet med andre konkurrenter. Kostnadseffektivitet er en av hovedfordelene ved å bruke Unity, og om man bare vil lage spill så er det helt gratis.

Ulemper:

- **Henger bak i verdensredigering** Det henger litt etter sammenlignet med andre verktøy som Unreal når det kommer til redigering av verden med terreng- og plasseringsverktøy. I utgangspunktet, når det kommer til et stort spill med store åpne verdener, vil vi ønske å være forberedt på mangel på assets tilgjengelighet og være forberedt på å gjøre mye mer arbeid på egen hånd.
- **Starte fra bunnen** Unity tilbyr ikke muligheten til å bygge spillet fra et fundament eller en mal. Selv om det gir et bredt bibliotek med verktøy i asset butikken, må vi lage spillet fra bunnen av på egen hånd.

3 Unreal

Unreal Engine er utviklet av Epic games og er brukt i mange typer industrier som et profesjonelt verktøy. Dette gjelder alt fra spill som FPS-er, AAA- spill og andre tredimensjonale spillsjangere til film og tv industrien. Som en spillmotor så er den blant de mest populære spill motorene på markedet og har et godt rykte for å ha laget noen solide spill som XCOM, Gears of war og Fortnite (Drake 2021). Spillmotoren er skrevet i C++ og støtter forskjellige plattformer som pc, mobil, konsoll og VR.

Fordeler:

- **Kan bruke .glTF og .obj filer** I basen av spillmotoren så er det to filtyper som kan importeres, .obj og .fbx filer. Dersom man ønsker å importere andre 3D modeller i andre filformater, så kan man bruke Unreal Datasmith. Unreal Datasmith er en kasse med verktøy for å forenkle prosessen av å flytte visuell data (objekter og 3D modeller) til Unreal. De har support for en del applikasjoner som Autodesk og SketchUp, og importer blant annet glTF filer, men ifølge dokumentasjonen så er den i en beta eller eksperimentell fase.
- **Virtuell Realitet** Unreal er svært egnet for VR utvikling på grunn av ytelsen som spillmotoren har. På grunna av dette så anbefales det å bruke VR utstyr som Gear VR, oculus og HTC Vive, som vil ligge i den dyrere klassen når det kommer til VR(N-iX 2018).
- **Lisensiering** Unreal tilbyr en gratis lisens (Creators licence), som gir mulighet for bruk av spillmotoren i bytte mot at prosjektet bare blir brukt internt i organisasjonen, og er et posjekt som ikke skal publiseres.
- **Visualisering og ytelse** Spillmotoren er en av de beste på markedet når det kommer til det visuelle og ytelse, noe som har mye å si hvis man har store prosjekter eller om man vil at spillet skal være realistisk og virkelighetsnær.
- **Blueprints** Unreal har blueprints noe som gjør at folk uten kjennskap til kode også kan gjøre enkle oppgaver, men den største og de mer avanserte bitene vil fortsatt bli programmert i C++.
- **Open source** Open source gjør at utvikling blir lettere og mer effektivt.

Ulemper:

- **Vanskelig å lære** Spillmotoren har visuell programmering noe som gjør at en som ikke kan programmering enklet kan lage noe enkelt som en demo, og er en funksjon som Unreal liker å pushe. For programmeringen så brukes det C++ som er sett på som ett vanskelig språk å lære (Roy 2021).
- **Ikke egnet for små grupper** Spillmotoren er ofte valgt av store korporasjoner med masse penger eller gode prosjekt grupper med masse erfaring, ettersom den ikke er godt egnet for små prosjekter og en manns grupper. Dette gjør at det kan være vanskelig å finne flinke folk til Unreal prosjekter (Eldad 2021).

4 Godot

Godot er en open-source community-driven game engine utgitt i 2014 som har vokst i bruk de siste årene, som er en del av FOSS. Den utvikles under MIT lisenset og har dermed "No strings attached, no royalties, nothing." som de skriver på hjemmesiden sin. Det er en engine som har utviklet seg til å bli feature-rich og kapabel til det meste fra 2D sidescrollers til 3D VR spill, og offisielt støtter Linux, macOS, Windows, Android, iOS, HTML5, og mer. På grunn av motoren sin open-source natur kan den utvikles til å støtte mer om det trengs, all koden er tilgjengelig og kan brukes helt fritt.

Godot er en kraftig spillmotor som er kapabel til det meste andre store spillmotorer som Unreal og Unity er, men det er usikkert hvor godt den egner seg til veldig grafisk intensive og store prosjekter da det ikke finnes mange store prosjekter som bruker Godot. Det vil sannsynligvis ikke være et problem for et prosjekt som dette, men det kan bety at optimalisering vil bli en prioritet om prosjektet blir veldig stort og grafisk intensivt. Godot bruker sitt eget scripting language "GDScript" som er bygget for å være enkel å lære, men støtter også C++ og C#.

Fordeler:

- **Lett å lære**, i følge Bryan Wirtz (Wirtz 2022), og vår gruppes oppfattelse av hva developer-samfunnet mener, i forumer o.l..
- **Virtuell Realitet**, God VR støtte med god dokumentasjon (Godot 2019)
- **God dokumentasjon**
- **Ofte oppdatert** av open-source fellesskapet

Ulemper:

- **Mindre hjelperessurser** enn større motorer. Godot har en mye mindre brukerbase og ble populært mye senere enn Unity og Unreal, og derfor er det vanskelig å finne støtte for mange problemer på nettet. Unity og Unreal har vært populære lenge nok til å sørge for at hvis du har et niche problem så finnes det hjelp for det på nettet.
- **Mindre asset store** enn Unity og Unreal, det finnes ikke like mange ferdiglagde moduler og plugins som i Unity og Unreal, så det finnes mindre hjelpemidler og programmet må sannsynligvis bygges fra bunnen av.

4.1 Open Source

Kildekode: https://github.com/godotengine/godot (Godot 2022a)

Det at Godot er open-source betyr at det utvikles av tusenvis av utviklere helt frivillig, under styret av to prosjektledere, dette har mange fordeler men også flere ulemper. Blant fordelene er det at som sagt er det no strings attached, det er ingen streng lisensavtale å følge, ingen royalties å betale, ingen krav, det er en helt fri spillmotor. Dette er ikke så veldig viktig for en utdanningsorganisasjon, men er fortsatt et pluss da andre motorer som Unity og Unreal kan ha forskjellige stipulasjoner på bruken av spillmotorene sine, og det er mulighet for forandring, mens Godot vil forbli fritt og open source. Siden kildekoden er åpen så kan motoren forandres etter brukerens behov så lenge de evner å programmere den, og den kan opprettholdes til all tid.

Fordeler:

- **"No strings attached"**, kan brukes helt fritt til hva som helst

- **Kan vedlikeholdes for alltid**, det er ingen fare for at firmaet som eier motoren legges ned og slutter støtten, fordi det er det online samfunnet som vedlikeholder den.
- **Kan forandres etter ønske**, hvis det er problemer med motoren eller nye funksjoner man ønsker å legge inn så er det bare å forandre kildekoden, men det krever kunnskap om C++ som motoren hovedsakelig skrives i.
- **Objektfiler** Motoren støtter objektfiler av typen .fbx og .obj.

Ulemper

- **Ikke offisiell støtte** på samme måte som Unreal og Unity gir, sidne organisasjonen som overser prosjektet er mindre og mer "hands off" er det ikke like mange muligheter for offisiell støtte om problemer oppstår, **man avhenger av community**.
- **Uforutsigbare forandringer** i framtidlige versjoner av Godot, siden det utvikles av samfunnet og ledes av to personer, "Project Manager Lead" Rémi Verschelde og "Technical and Project" Juan Linietsky (Godot 2022b), som har hele ansvaret for merging av kode i den offisielle hoved-grenen av Godot.
- **Usikker offisiell støtte i framtiden**, Godot organisasjonen drives av månedlige donasjoner som for øyeblikket fungerer kjempegodt, men det betyr at den offisielle støtten som finnes er usikker.

Det er mange fordeler med åpen kildekode, men gitt vår antagelse av at lokførerskolen ikke ønsker å forandre kildekoden til motoren, og ikke har behov for et komplett fritt lisens, så mener vi at fordelene ikke er like relevante for lokførerskolen som ulempene; mangelen på offisiell støtte og internett-ressurser relativt til Unity og Unreal.

5 CryEngine

Det som gjør CryEngine verdig på listen, er dens grafiske evner som overstråler Unitys grafiske evner og tilsvarer det Unreal har. Selv om det er en tung og kraftig spillmotor, bruker CryEngine litt tid på å kunne bruke denne plattformen effektivt og litt vanskeligere å forstå for nybegynnere som ikke har brukt noen andre spillmotorer på forhånd. CryEngine løper på C++ programmeringsspråket og oppstod først som en Sandbox editor utviklet av CryTek. (CryTek 2022)

Fordeler:

- **Kan bruke .glTF og .obj filer** Motoren støtter også imortering av objektfieler i blandt annet .fbx og .obj.
- **Virtuell Realitet** CryEngine er kjent for å støtte Playstation VR, HTC Vive og Oculus Rift.
- **Høy grafikk med god ytelse** CryEngine har vært årevis kjent for grafiske høy gjengivelse etter å ha drevet Crysis- og FarCry-seriene.
- **Bra håndtering av åpen verden** CryEngine håndterer åpen verden bra. CryEngine kan betraktes som en pioner når det gjelder å skape høy visuell troskap.
- **Lav kostnad** CryEngine koster 9.90 EUR/USD per måned og tar 5% royalties. (Steam 2022)

Ulemper:

- **Hovedsakelig bygd for kun spill** Verktøyene som CryEngine har er bra og veldig relevant til spillutvikling, men ikke for eksempel mobilutvikling eller simulasjoner.
- **Mangel på ressurs og dokumentasjon** Mangel på ressurs og dokumentasjon henger godt sammen. Dokumentasjon er svake, i hvertfall for nybegynnere, og spillmotoren er ikke særlig populær blant utviklere og dermed er det lite tutorials eller kurs fra eksterne kilder.
- **Ikke for nybegynnere** Vi har ikke hatt erfaring med CryEngine i forhånd og dermed blir det mer utfordrende å levere en produkt som tilfredsstiller alle kriteriene i en kort tidsramme. Det er også derfor det er vanskelig for utviklere å få noe godt ut for prosjekter i virtuell realitet.

6 Konklusjon

Etter å ha fullført analysen vår så står vi igjen med disse resultatene som gjenspeiler de kravene som er ønsket oppfylt av lokførerskolen:

Game Engine	Gjenbruke resursser	Lett å lære	Joystick support
Unreal	x		x
CryEngine	x		x
Unity	x	x	x
Godot	x	x	x

Under vår undersøkelse av spillmotorene så har vi funnet ut at det er mye som kommer fram som subjektivt mår det kommer til valg av spillmotor. Spesielt under undersøkelsen av om en motor er lett å lære eller ikke. Derimot så har vi basert ratingen på det vi fikk inntrykk av da vi leste på forskjellige meningsmålinger og artikler. Vi har også tidligere erfaring med Unity og litt Unreal så vi har også lagt litt personlige meninger i dette punktet. Ellers så har vi også sett på forskjellige ulemper med motorene.

Med dette så har vi bestemt oss for å bruke Unity, og noen grunner for dette er følgende. Unity oppfyller alle de ønskede kravene til lokførerskolen. Det gjør også Godot men vi mener at den spillmotoren er ustabil ettersom den når som helst kan miste offisiell støtte og er bygget av et felleskap og ikke av en organisasjon. Unity er også lett å lære, noe som gjør det enkelt å få inn folk for videreutvikling etter at bachelor oppgaven er ferdig. Vi har mye erfaring med Unity fra før av, noe som gjør at vi kan starte tidlig.

Bibliography

N-iX (2018). *Unity vs. Unreal: How to Choose the Best Game Engine.* URL: https://medium.com/@N_iX/unity-vs-unreal-how-to-choose-the-best-game-engine-d3dbb4add73c. (Besøkt 22.01.2022).

Godot (2019). *Godot - Vr Starter Tutorial.* URL: https://docs.godotengine.org/en/3.1/tutorials/vr/vr_starter_tutorial.html. (Besøkt 17.01.2022).

Perforce (2020). *Unity vs Unreal, What Kind of Game Dev Are You?* URL: https://www.perforce.com/blog/vcs/most-popular-game-engines. (Besøkt 20.02.2022).

Drake, Jeff (2021). *20 Great Games That Use The Unreal 4 Game Engine.* URL: https://www.thegamer.com/great-games-use-unreal-4-game-engine/. (Besøkt 21.01.2022).

Eldad, Asaf (2021). *What Are the Most Popular Game Engines?* URL: https://www.incredibuild.com/blog/unity-vs-unreal-what-kind-of-game-dev-are-you. (Besøkt 22.01.2022).

Roy, Ashmita (2021). *The Hardest and Easiest Programming Languages to Learn for FAANG Interviews.* URL: https://www.interviewkickstart.com/blog/hardest-and-easiest-programming-languages-to-learn. (Besøkt 22.01.2022).

CryTek (2022). *Crytek Hjemmeside.* URL: https://www.crytek.com. (Besøkt 23.01.2022).

Godot (2022a). *Github - Godot Kildekode.* URL: https://github.com/godotengine/godot. (Besøkt 17.01.2022).

— (2022b). *Godot Hjemmeside.* URL: https://godotengine.org. (Besøkt 21.01.2022).

Steam (2022). *CryTek Pricing on steam.* URL: https://store.steampowered.com/app/220980/CRYENGINE/. (Besøkt 23.01.2022).

Unity (2022a). *Unity Documentation 2021.* URL: https://docs.unity3d.com/2021.2/Documentation/Manual/index.html. (Besøkt 17.01.2022).

— (2022b). *Unity Powers VR Training.* URL: https://unity.com/our-company/newsroom/iitsec-unity-powers-3d-virtual-training-simulations-defense-military-aerospace. (Besøkt 17.01.2022).

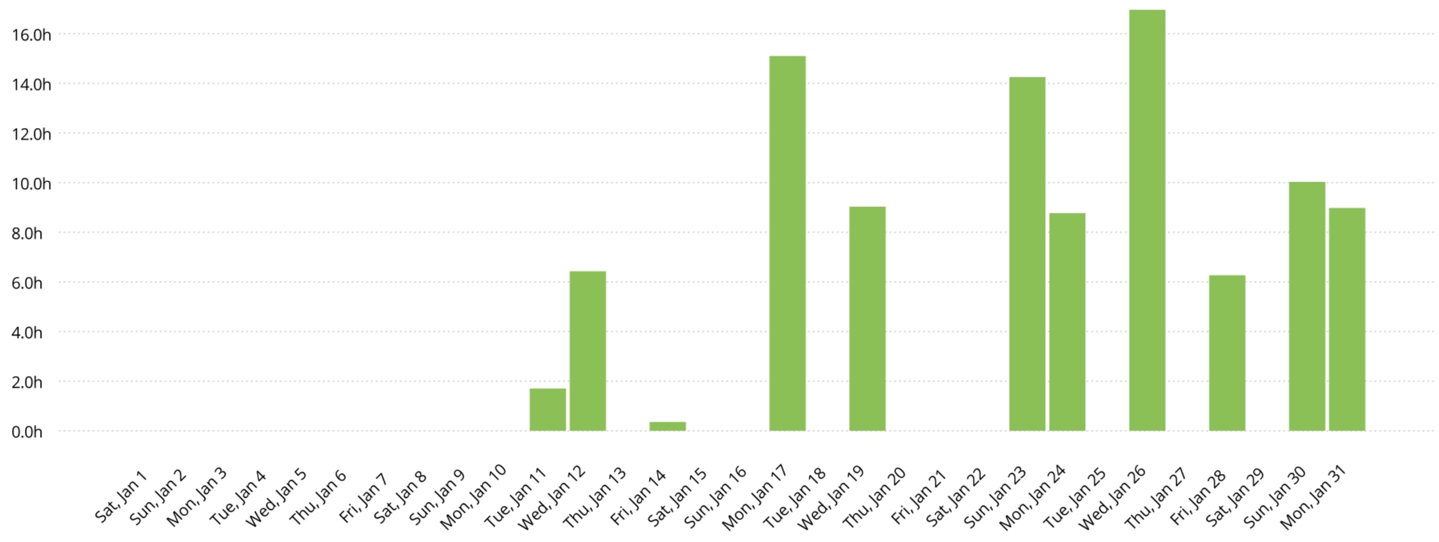
Wirtz, Bryan (2022). *GameDesigning.org - Unity vs Godot.* URL: www.gamedesigning.org/engines/unity-vs-godot/. (Besøkt 17.01.2022).

E Appendix E

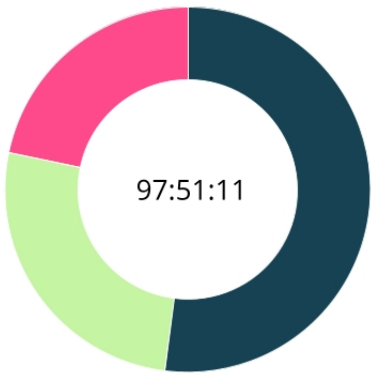
Summary report

01/01/2022 - 31/01/2022

Total: 97:51:11 Billable: 97:51:11 Amount: 0.00 USD



User



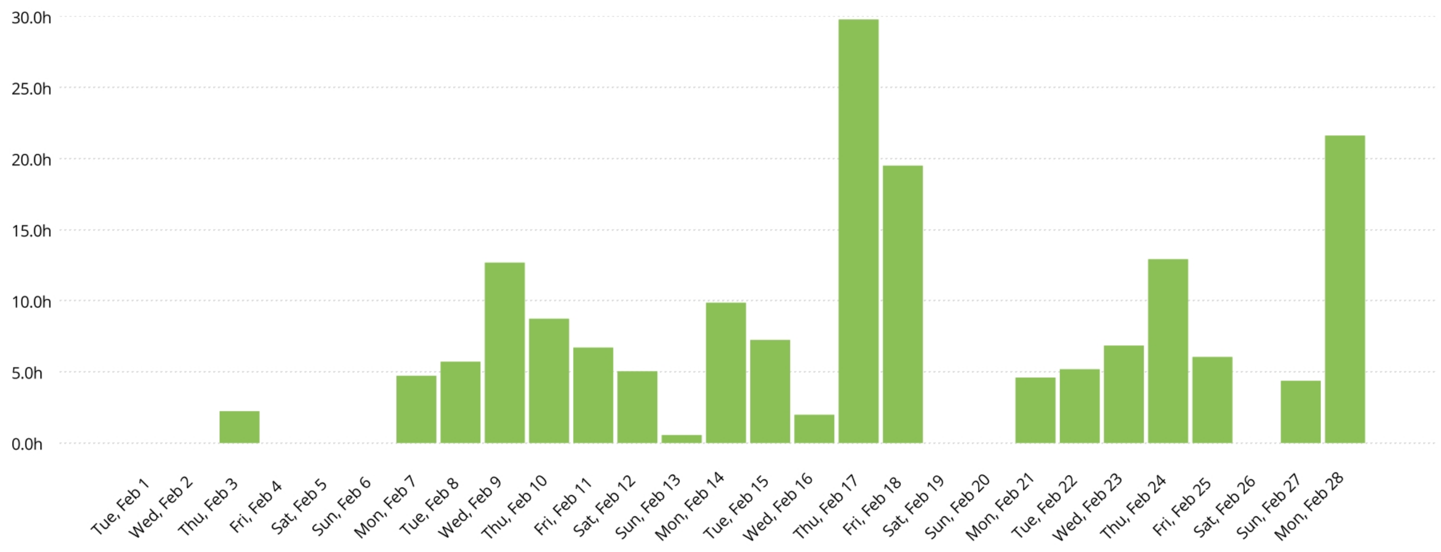
●	Julian Kragset	21:18:01	21.77%
●	MichaelK	25:41:15	26.25%
●	Piocram	50:51:55	51.99%

User	Duration	Amount
Julian Kragset	21:18:01	0.00 USD
MichaelK	25:41:15	0.00 USD
Piocram	50:51:55	0.00 USD

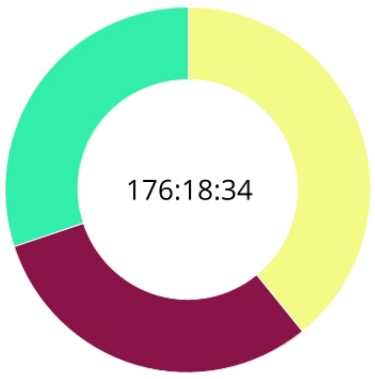
Summary report

01/02/2022 - 28/02/2022

Total: 176:18:34 Billable: 176:18:34 Amount: 0.00 USD



User



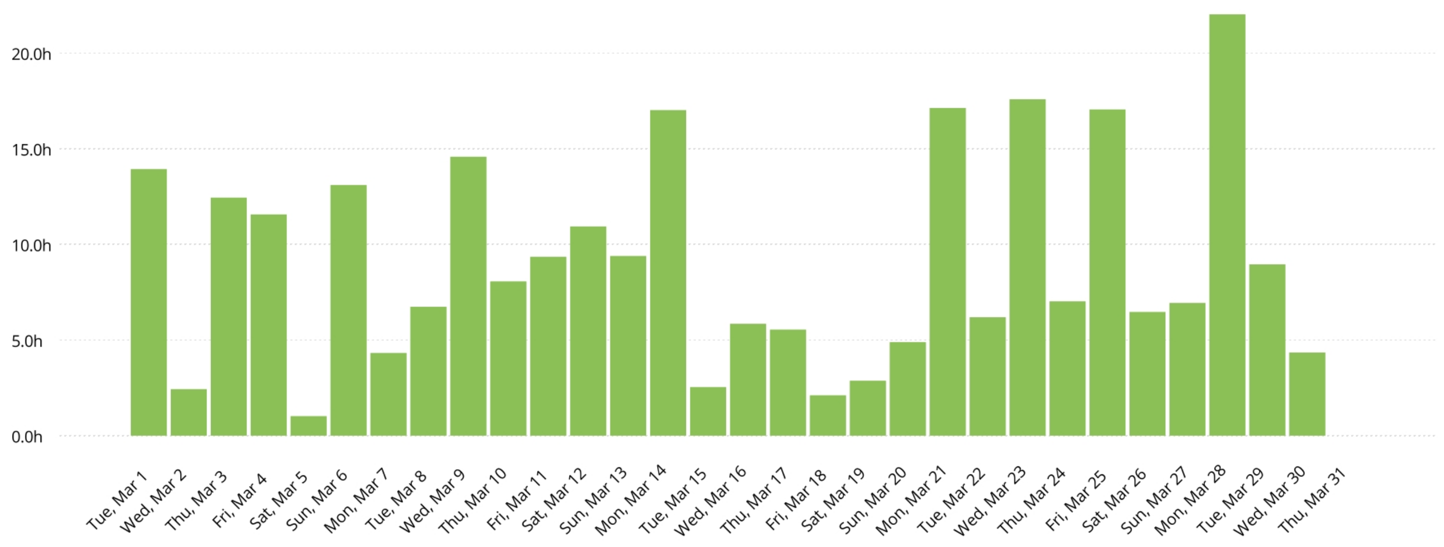
●	Julian Kragset	52:57:48	30.04%
●	MichaelK	54:19:26	30.81%
●	Piocram	69:01:20	39.15%

User	Duration	Amount
Julian Kragset	52:57:48	0.00 USD
MichaelK	54:19:26	0.00 USD
Piocram	69:01:20	0.00 USD

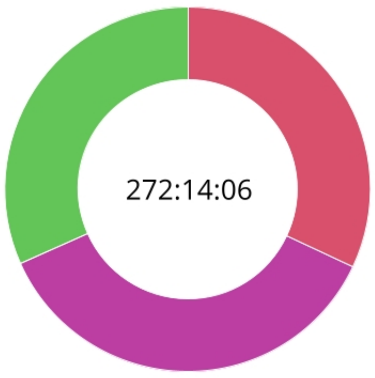
Summary report

01/03/2022 - 31/03/2022

Total: 272:14:06 Billable: 272:14:06 Amount: 0.00 USD



User



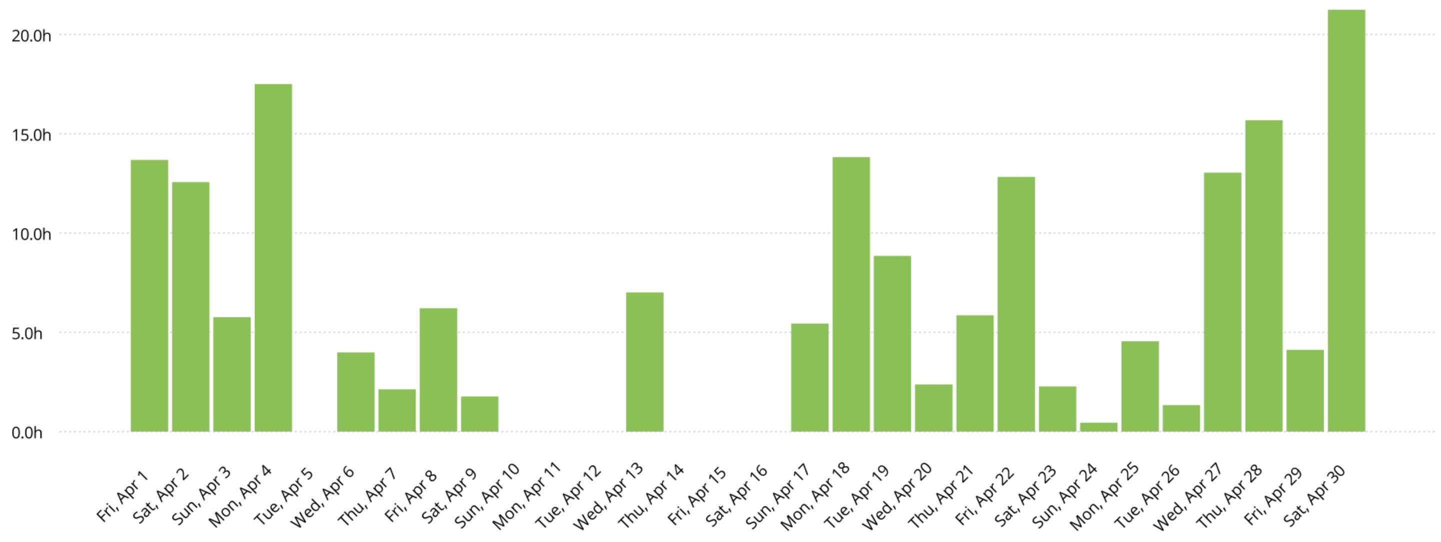
●	Julian Kragset	86:15:33	31.69%
●	MichaelK	98:52:24	36.32%
●	Piocram	87:06:09	31.99%

User	Duration	Amount
Julian Kragset	86:15:33	0.00 USD
MichaelK	98:52:24	0.00 USD
Piocram	87:06:09	0.00 USD

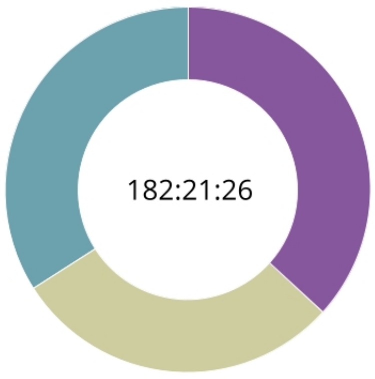
Summary report

01/04/2022 - 30/04/2022

Total: 182:21:26 Billable: 182:21:26 Amount: 0.00 USD



User



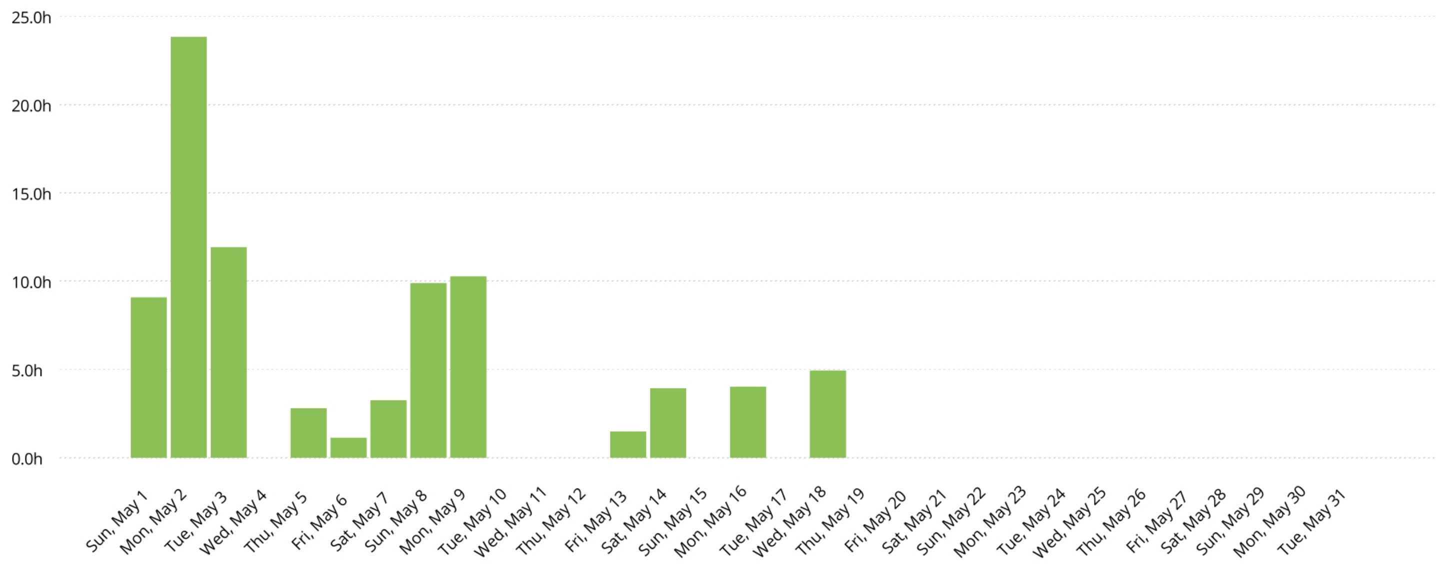
●	Julian Kragset	62:04:43	34.04%
●	MichaelK	53:20:09	29.25%
●	Piocram	66:56:34	36.71%

User	Duration	Amount
Julian Kragset	62:04:43	0.00 USD
MichaelK	53:20:09	0.00 USD
Piocram	66:56:34	0.00 USD

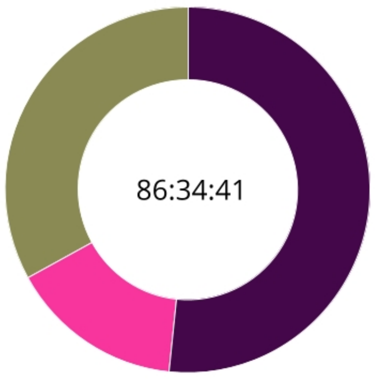
Summary report

01/05/2022 - 31/05/2022

Total: 86:34:41 Billable: 86:34:41 Amount: 0.00 USD



User



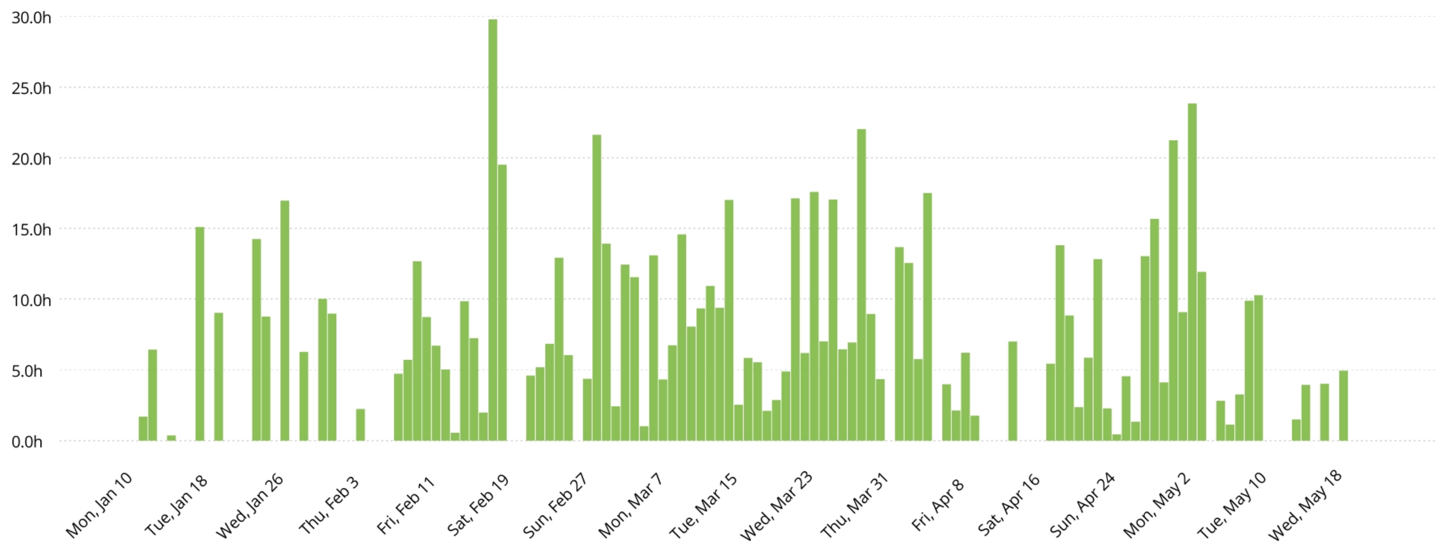
●	Julian Kragset	28:33:45	32.99%
●	MichaelK	13:18:31	15.37%
●	Piocram	44:42:25	51.64%

User	Duration	Amount
Julian Kragset	28:33:45	0.00 USD
MichaelK	13:18:31	0.00 USD
Piocram	44:42:25	0.00 USD

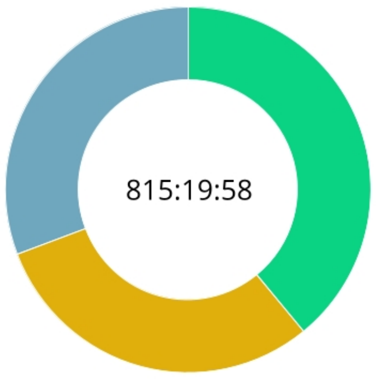
Summary report

10/01/2022 - 20/05/2022

Total: 815:19:58 Billable: 815:19:58 Amount: 0.00 USD



User



●	Julian Kragset	251:09:50	30.80%
●	MichaelK	245:31:45	30.11%
●	Piocram	318:38:23	39.08%

User	Duration	Amount
Julian Kragset	251:09:50	0.00 USD
MichaelK	245:31:45	0.00 USD
Piocram	318:38:23	0.00 USD

F Appendix F

Useful Links

For building and setting up a new scenario:
https://youtu.be/hLDLBRX8o2M

MapMagic2 tutorial:
https://www.youtube.com/watch?v=-fO8P-dp28E&list=PL8fjbXLqBxvbsJ56kskwA2tWziQx3G05m

GitHub Repository:
https://github.com/Bachelor-Group-203/Desksim

Final Demo build:
https://github.com/Bachelor-Group-203/Desksim/releases/tag/release