Bachelor's thesis

Information technology

NTIETS13P

2017

Kasperi Ekqvist

IMPLEMENTING USER INTERFACE FUNCTIONALITY FOR MOBILE GAMES IN UNITY GAME ENGINE

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information Technology | Game Technology

June 2017 | 64 pages

Instructor: Lecturer Werner Ravyse

Kasperi Ekqvist

IMPLEMENTING USER INTERFACE FUNCTIONALITY FOR MOBILE GAMES IN UNITY GAME ENGINE

The aim of this thesis was to form a set of guidelines and best practices for both implementing and optimizing user interface functionality developed in the Unity game engine. This was achieved with the implementation and the optimization of user interface functionality for two separate mobile games, Movenator and Mini Golf Universe.

For both projects, the initial requirements were set and the final products were developed based on the required functionality. The number of interfaces that were developed and later examined was six for Movenator and four for Mini Golf Universe. This thesis describes both final products before examining the technical implementations in more detail. In the optimization process, two specific optimization tools were used, a custom framerate counter and the Profiler of the Unity game engine.

A table of the development choices was compiled to present the findings. In this table, the choices that were made for the two games were compared, and conclusions were drawn based on those. This involved analyzing both the similarities and the differences of the choices, as well as the reasoning behind the choices. Another table was compiled that shows the effects of optimization. This included data of the number of draw calls made and the framerate that was recorded both before optimization and after the process.

Based on this data, a set of guidelines and best practices were defined for Canvas objects, Text objects, object positioning for both static and dynamically added objects, Scroll View objects, and functionality scripting. Both the custom framerate counter and the Profiler were determined to be useful for the optimization process.

KEYWORDS:

Unity, UI, mobile, games, optimization

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietotekniikka | Peliteknologia

June 2017 | 64 sivua

Ohjaaja: Lecturer Werner Ravyse

Kasperi Ekqvist

MOBIILIPELIEN KÄYTTÖLIITTYMÄTOIMINNALLISUUDEN TOTEUTTAMINEN UNITY-PELIMOOTTORILLA

Opinnäytetyön tavoitteena oli muodostaa suosituksia Unity-pelimoottorilla luotujen käyttöliittymätoiminnallisuuksien toteutusta ja optimointia varten. Tämä tehtiin toteuttamalla ja optimoimalla käyttöliittymätoiminallisuudet kahteen eri peliin. Pelit olivat Movenator ja Mini Golf Universe.

Molempiin projekteihin asetettiin alkuvaatimukset, ja lopulliset tuotokset kehitettiin vaaditun toiminnallisuuden mukaisesti. Lopputuotteita kuvailtiin ensin kokonaisuuksina ja tämän jälkeen syvennyttiin niiden teknisiin toteutuksiin tarkemmin. Optimointiprosessissa käytettiin erityisesti kahta eri optimointityökalua, joita olivat pelimoottorin oma Profiler-työkalu ja erikseen integroitu ruudunpäivitysnopeuslaskuri. Tuloksiksi saatiin prototyyppikäyttöliittymät molempia projekteja varten. Käyttöliittymäkokonaisuuksia toteutettiin Movenator-peliin kuusi ja Mini Golf Universe -peliin neljä.

Tämän jälkeen tarkasteltiin molempien pelien kehityksessä tehtyjä päätöksiä ja päätöksien yhteneväisyyksiä, eriäväisyyksien ja näihin johtaneita syitä. Lisäksi tarkasteltiin, millä tavoin optimointia oli toteutettu. Näiden tietojen perusteella suosituksia muodostettiin Canvas-objekteille, Text-objekteille, dynaamisesti lisättyjen ja staattisen objektien sijoittelulle, Scroll View -objekteille ja toiminnallisuuden skriptaamiselle. Sekä integroitu ruudunpäivitysnopeuslaskuri että pelimoottorin Profiler-työkalu todettiin hyödyllisiksi optimointiprosessin kannalta.

ASIASANAT:

Unity, käyttöliittymä, mobiili, pelit, optimointi

CONTENT

PICTURES

TABLES

LIST OF ABBREVIATIONS

2D	Two-dimensional
3D	Three-dimensional
API	Application programming interface
CPU	Central processing unit
FPS	Frames per second
GPU	Graphics processing unit
UI	User interface

1 INTRODUCTION

The global games market is continuously growing, and mobile games are expected to take over half of the market by 2020 (Brightman 2017). Even on Apple's App Store alone, the generated revenue has exceeded 70 billion dollars, the number of downloaded applications has grown by 70 percent in the last 12 months, and the category of games and entertainment is the most popular (Dring 2017). Both the supply and the demand of mobile games is growing all the time.

For this thesis, I worked on two separate mobile games. Both games were developed using the Unity game engine; an engine that I was already previously familiar with. It is also an engine that is the most popular third-party game engine among the top 1000 free mobile games (Unity Technologies 2017a).

I developed user interface (UI) functionality for two different mobile games, Movenator and Mini Golf Universe, with the aim of producing a set of guidelines of best practices for the implementation and optimization of UI functionality for mobile games in general; this was done specifically in the Unity game engine environment.

The Movenator project was a part of a research project at the Department of Nursing Science at the University of Turku; its purpose "is to promote the physical activity (PA) and PA self-efficacy of pre-adolescents" (University of Turku 2017). The development of the current prototype started in November 2016, and the development of its UI functionality started in early 2017.

Mini Golf Universe was a game project of a local game start-up, which was made for pure entertainment purposes. The development of it started in early 2017, and the development of its UI functionality started in May 2017.

1.1 Unity Game Engine

Unity offers the developers a wide variety of tools that can be used in the process of developing and implementing a UI for a game or an application, but it is up to the developer to do so efficiently. The fact that some of the pre-packaged solutions are not well-optimized increases the difficulty of making the right decisions (Tantzy Games 2016).

Unity is a popular game engine developed by Unity Technologies. Since the launch of its first version in 2005, Unity has grown to become one of the most used game development software in the world (Fear 2009; Unity Technologies 2017a).

The story of Unity started in 2002 when two of the original founders, Nicholas Francis and Joachim Ante, of Unity met each other through a mailing list while working on their own game engines. After some time, they decided to combine their forces to develop one single game engine which would later be known as Unity. (Fear 2009.)

The original intention of the founders was to develop a game engine that they could use to not only license to others but also use to develop games themselves. However, the focus later shifted to only developing a game engine for others to use. (Fear 2009.)

The Unity software was also originally only available for use on the Mac platform. The development team had to build it again from the ground up to support the Windows platform. The first version with Windows support was not released until 2009. (Fear 2009.)

In the first quarter of 2016, 34 percent of the top 1000 free mobile games were developed with Unity, more than with any other third-party game development software. In the third quarter of 2016 alone, 5 billion games developed with Unity were downloaded. Furthermore, games developed with Unity have reached 2.4 billion unique mobile devices. (Unity Technologies 2017a.)

There are multiple reasons why Unity is the software of choice for both Movenator and Mini Golf Universe. Firstly, it is free to use for individuals and companies whose annual revenue is less than $100,000, and furthermore, it is completely royalty free (Unity Technologies 2017b). For organizations, such as universities, and small game development companies, it is an excellent choice because it does not place as a great of a financial strain on the developer, and therefore, it also reduces risk for entering the market.

Unity also features a multiplatform support. This allows developers to quickly deploy the game to multiple different platforms. The engine features support for a wide variety of mobile, virtual reality, augmented reality, desktop, console, web, and smart TV platforms. (Unity Technologies 2017c.).

Since Movenator and Mini Golf Universe are set to be released on mobile devices, the most attractive feature of Unity is its mobile support for both iOS, the operating system of Apple mobile devices, and Android, the operating system used by most other mobile device manufacturers (Unity Technologies 2017c). According to data provided by the American market research company International Data Corporation (2017), Android and iOS smartphones formed a total of 99.3 percent of the smartphone market in the third quarter of 2016, based on unit shipments.

Furthermore, Unity features a comprehensive user manual and a documentation of the scripting application programming interface (API) to support developers working with the software. The user manual “helps you learn how to use the Unity Editor and its associated services” (Unity Technologies 2017d).

The Scripting API provides developers “details of the scripting API that Unity provides”; the Scripting API features example code for developers to study to gain a better understanding of the API, and it offers a lot of these examples in both C# and a heavily modified version of JavaScript. (Unity Technologies 2017e.)

It should be noted that Unreal Engine is another well-known option for 3D game development. Like Unity, it is free to use, but unlike Unity, the developers “are obligated to pay to Epic 5% of all gross revenue after the first $3,000 per game or application per calendar quarter” (Unreal Engine 2017). Unreal Engine also supports Android and iOS mobile platforms. However, free access to the Unity game engine has been around for a much longer period of time. Therefore, there are a lot more tutorials and documentation available for Unity, and that is why Unity is still much more widely used in game development, especially when targeting mobile platforms.

1.2 Optimization

To start the work of optimization in practice, or to even discuss it further in detail, some context must be provided. It must first be known what exactly is meant by optimization and what is meant by it in the context of video game development. It should also be clear how important optimization is, and in which circumstances it is important.

When optimization is exercised correctly, it leads to improvements in the game’s performance. This performance consists of the performance of all the various parts of hardware, such as the central processing unit (CPU), the graphics processing unit (GPU), and the memory. Optimization can help ease the burden on the hardware, increasing the performance, and providing a better game experience for the player.

Optimization itself is a concept that has been used for a long time. It does not have a concrete meaning without any context or explanation. However, what is generally meant by optimization in the context of video games, is the act of assuring that the performance of a game is on the best level it can be on a wide range of different target platforms. Even though the importance of optimization is commonly known, it is still not practiced to the extent that it requires. There are three different key issues. (Thorn 2013, 259.)

Firstly, optimization in game development is not something that should be done only when the development is done. Optimization is not synonymous with the concept of polish which only happens as the game in development nears completion. To only consider optimization at this stage, can be disastrous. Optimization should already be a part of the design process of the game. (Thorn 2013, 259.)

Secondly, optimization cannot be done using the exact same approaches and methods for all games. Every part of optimization does not apply for every game in development. It is required to identify the areas of the game in question that could be and should be optimized. Optimization can “require lots of hard work and planning”, but at times it “can be made by making only the simplest of adjustments”. (Thorn 2013, 260.)

Finally, while optimization is more important when dealing with mobile devices, due to their relatively more limited power level, it should be noted that optimization is equally necessary on all platforms. There is no platform that would have access to unlimited processing power. Thus, games of larger scale, or of poor execution, always have the possibility to reach the limits of the hardware it is run on. (Thorn 2013, 260.)

2 RESEARCH

2.1 Problem Statement

Developers should aim to properly optimize the UI implementations and functionality in games. Naturally, they should aim to do the same for all aspects of game development. However, the UI is rarely the focus of the gameplay experience, but rather something that, when properly implemented, should help the player enjoy the actual game content. For that reason, the UI should be implemented and optimized so that it does not become a burden on the performance.

A UI that performs poorly can diminish the game experience. For the user, playing a game and using an interface which performs in a sluggish or unresponsive manner can induce feelings of frustration or even anger with the game or application; this could decrease the user’s desire to play the game or interact with portions of it which naturally is never the intention behind the development process.

This is especially true for mobile games. When games are run on desktops, laptops, consoles, or other relatively powerful machines, the developers and the consumers of video games have access to much more power than on mobile devices. Even game features which seem fairly usual can easily clog the performance of a mobile device; these features can be as simple as the initialization of objects in the game world, be they characters or a part of the UI.

Nowadays, a lot of the newer mobile devices pack a lot of punch. However, as a game developer, all the different devices in use must be considered already during the development process. This allows the final product to be playable for as many customers as possible. Older and less expensive devices usually perform worse; all the functionality might not work properly on them, especially if it has been implemented with care.

However, implementing a UI with all its functionality for a game is not necessary. The Unity game engine offers a wide variety of tools to implement UI elements for games. The problem for a developer can be to choose the ones that best fit the task at hand.

The Department of Nursing Science at the University of Turku has already been involved in the development of different game projects, and they will most probably continue to do so as we move towards a world where technology and gamified elements are getting more and more popular. They also occasionally employ students, who can still be inexperienced in the field of game development, for their projects. Because of this inexperience, game interface optimization is often overlooked until the diminished performance is noted during the play-test phase. Attempting to rectify the optimization issue at this stage results in reworking much of the coding and even altering designs. In money and time terms, this is a costly oversight that should be avoided.

By following a simple set of guidelines and best practices, even less experienced developers can successfully implement a UI that does not hinder the game experience, but rather supports it. The guidelines formed as a part of this thesis could help these less experienced developers to get a better understanding of how different UI tools in the Unity game engine work and what they can be used for, without consulting different multiple sources.

2.2 Aim

The aim of the thesis is to produce a set of guidelines for best practices for implementing UIs for mobile games in the Unity game engine. These guidelines will not cover everything that can be done with the UI tools, but rather focus on the areas that will be encountered during the development of two separate game projects introduced earlier. These areas include the kind of functionality that is arguably most commonly used in different game projects, and thus forms the basis of UIs.

Only general guidelines for implementing UIs will be set. If the guidelines and best practices set here are too game-specific, they might be of no use when implementing UI functionality for other games, even if the work was done using the same game engine.

2.3 Objectives

To produce a set of guidelines for UI implementation and optimization, this thesis will explain a set of Unity-based UI tools and describe how to use them and how to assess their effect on optimization so that the right interface tools are given the highest priority according to the impact they may have on optimization. Furthermore, this thesis will investigate the different optimization tools offered by Unity and describe the thought process that goes into deciding which tools are most appropriate for the task at hand.

The first objective is to not only list and discuss the various interface tools, but also to include key aspects that need to be considered in the implementing of efficient functionality for the UI elements to interact with both the user as well as other game elements and UI elements.

The second objective is the comparison and the purposing the optimization tools offered by Unity. There are two main tools that are the focus of this study; they are Profiler, which provides general information on performance and resource allocation, and Frame Debugger, which provides additional information and statistics on the graphics rendering process. Additionally, the functionality and usefulness of a custom frames per second (FPS) counter, that has been integrated into the game engine, will be assessed.

2.4 Methods

The main method of completing the objectives, that have been set for the thesis, is by a detailed development journal. This journal will offer insight to the choices that have been made regarding the UI tools that have been chosen to be used, and the ways that different pieces of functionality have been implemented.

The focus of the journal will be Movenator, the game project of the Department of Nursing Science at the University of Turku. This is where the bulk of the work behind the thesis lies. However, additional insights from Mini Golf Universe, a separate game project will be offered as well. This is done to provide more basis for setting up guidelines for the best practices, and to prevent the guidelines, which should be fairly generic, from being based exclusively on experience from a single game project and specific circumstances.

Naturally, as also mentioned before, this Bachelor’s thesis will not cover every possible aspect of implementing UIs in the Unity game engine. However, a journal of the development process of the basic aspects of UIs for the two different games should provide a clear documentation of the process and the guidelines.

Both games in question are to be released on mobile platforms in the future. Movenator will be focused on elements such as building towns and competing in match-3 puzzles. On the other hand, as the name suggests, Mini Golf Universe is a minigolf game. This means that the actual gameplay goals of the games are very different. That is why they are good choices for the process of determining a single set of universal rules for the development of UIs for mobile games.

The idea is to find the optimized and simplified solutions to various aspects of the UI implementation. When analyzing and comparing the solutions that are developed for each of the games, it is important to try to examine the solutions in a general context, rather than in a game-specific context. Naturally, each game developed is a separate entity, and thus requires its own solutions to its specific problems and needs. However, it is important, for the sake of the guidelines, to determine the practices that are generally the most useful.

In this thesis, documenting the thought and development process is how the data is collected. The documentation of the process should be detailed enough so the reader can easily follow the process. The data that is collected should have clear examples of different implementations, and offer context of their use. This should create, in certain cases but not necessarily all of them, a timeline of the development process of a certain functionality. This allows the reader to also gain a better understanding of the though process behind the development and determination. Based on the collected data, some of the best practices for developing UIs in the Unity game engine can be determined by analyzing the data.

When the data is analyzed, the practices that are presented as optimal, or even the best in certain situations, should also always offer reasoning as to why they have been analyzed and determined to be just that. The reasoning behind these decisions and determinations can be based on several different forms of analysis; these include things like measured efficiency, graphical determination, or various sources. The analysis also might not always be limited to simply one of these, but can include more of them.

Firstly, the analysis can be conducted by measuring the efficiency of different solutions. For this thesis, there are several ways for measuring the efficiency of an implementations or a functionality. Some of these are packaged with the Unity game engine itself. These include the Profiler and the Frame Debugger windows. There are also additional tools that can be developed and used. The usefulness of the different optimization tools depends on the matter at hand; some are naturally better and more useful at certain aspects than others, and vice versa.

The Profiler is a general optimization tool; it provides information on the amount of resources spent on various aspects of the game, from rendering the graphics to executing scripts. It also provides information on what kind of resources are used for different processes; these can include the GPU, the CPU, and memory of the computer.

Should the Profiler be used in the optimization process, the focus should generally be on “those parts of the game that consume the most time”. The Profiler should also be used both before and after making changes; this is done to ensure that the changes made have been beneficial to the performance and not vice versa. (Unity Technologies 2017f.)

The Frame Debugger is used to, as the name implies, analyze single frames of the game. The tool can be used to pause the game and gain information on how a specific frame of the game is rendered; this is done to gain a better understanding how all the graphics elements are drawn on the screen. The Frame Debugger provides information on the draw calls, in which the CPU sends data to the GPU to render on the screen for the user, and their total number; this can help the user to identify problem areas in the rendering process. There could be an unnecessarily high amount of draw calls for objects for which the process could be simplified; there could also be simply any number of draw calls for objects which are not even displayed for the player, and therefore may not even need to be rendered. (Unity Technologies 2017g.)

In addition to the two tools that come prepackaged with the game engine, a custom tool should be integrated to measure the framerate, i.e. the FPS being rendered, of the games that are developed; this is done because the framerate provided by the Profiler is not entirely accurate. The Profiler calculates the FPS by dividing 1 by the amount of time, usually only milliseconds, the CPU spent on that specific frame. To measure the framerate more accurately, the actual number of frames being rendered each second should be calculated. (Flick 2015.)

The use of these different tools as a part of the process of determining the optimal implementations also helps in completing the objective of determining their own usefulness in the process. The occasions that these tools are used as a part of the determining process will then be further analyzed as a part of the results.

Secondly, the analysis can be simply based on the visual aspects. These can come into question when, for example, a graphical element of the UI needs to be anchored onto the screen in a certain way. This might be required because of the need to support a wide variety of different devices, and a certain element might always be required to, for example, appear in a certain part of the screen area.

To back up this form of analysis, the UI tools and their specific elements should be carefully explained. In practice, screenshots of the situation at hand should be presented to offer the reader a better understanding of the results of an analysis of this kind.

In summary, the research method will journal the game development of two games; make a comparison of the journals; extract the common decision-making processes from both journals; show the performance impact that the optimization decisions have had; and produce a set of guidelines for UI implementation and optimization.

3 REQUIREMENTS AND FINAL PRODUCTS

Different sets of requirements were set for each of the projects. These acted as guidelines for the UI elements and functionality that were to be developed. For the Movenator project, a lot of UI functionality was required; this functionality was mainly built around the part of the game that focused on city-building. For Mini Golf Universe, the task was to build the basic UI functionality of its main menu.

In this section, the focus will be on the various parts of the UIs of the games developed. The initial requirements will be provided, and the final implementations will be discussed briefly to give a better understanding of the end result. However, the decisions made during the development and optimization process for each of the elements will be discussed more in detail in the next section.

3.1 Movenator

The task for the Movenator project was to build most of the UI functionality that was required for the player to play the city-building part of the game; this involved many kinds of menus, panels, and interfaces. A user experience designer also worked on the project. His responsibility was to design the look and most of the general functionality of the UI; he set the basic requirements for the kind of functionality that was needed to be developed.

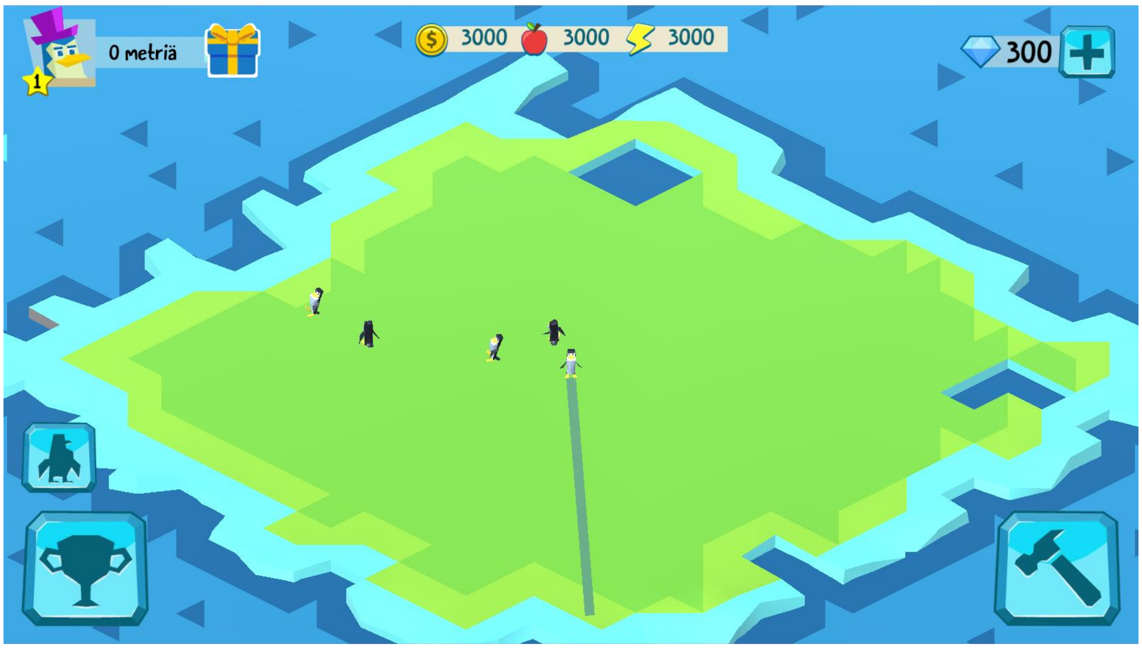
The designer provided plans of all the various parts of the UI; the development was based on his plans. It should be noted that his plans involved the graphical look and the functionality that was required for the project, but he was not in charge of the actual technical implementation.

The development process was spread over several different months; during the development, the designer of the user experience also conducted tests where people tested a wireframe version, i.e. a visual representation of the future UI and its functionality. This provided valuable feedback, but also caused some significant changes to functionalities that had already been implemented, and thus, slowing down the overall rate of progression.

For each of the different UI elements, the final products will be presented. Both the initial requirements and the final implementations will be discussed to provide more context on how they function.

3.1.1 General Game Interface

As the game is focused on city-building, there were several traditional elements that wanted to be shown in the default city view. These elements included different buttons as well as relevant information that needed to be displayed for the player; these can all be seen in Picture 1.



Picture 1. General game interface for Movenator.

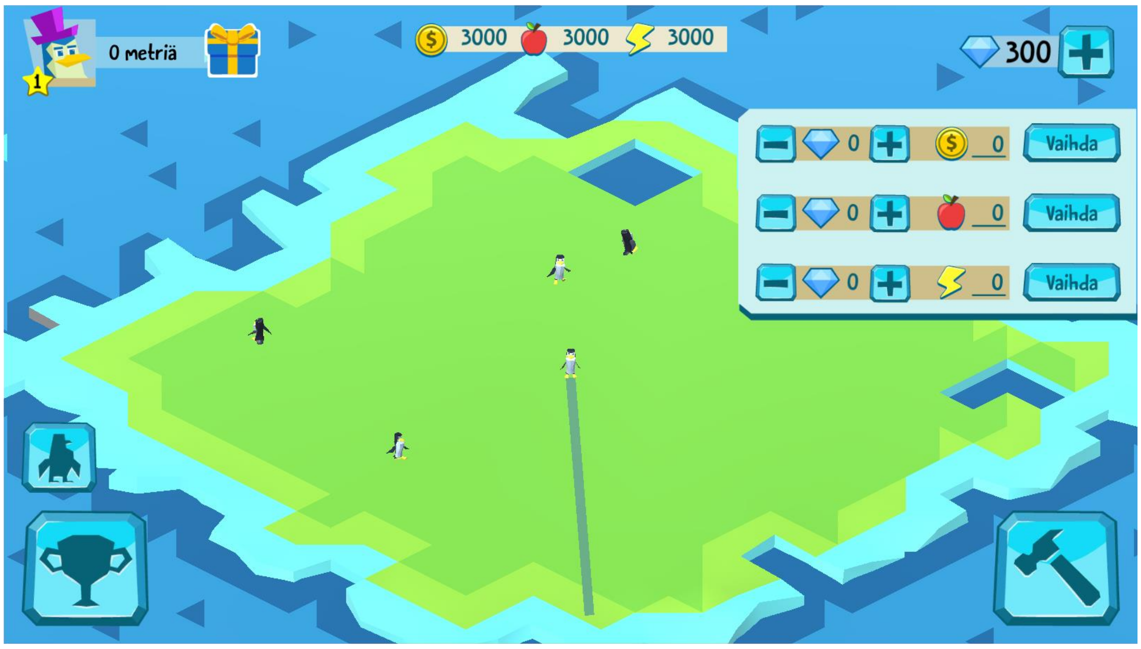
Buttons were included for opening all the game's other interfaces, and all the buttons featured a specific image to symbolize its functionality; a plus sign for the resource exchange interface, a penguin for the team interface, a trophy for the competition interface, and a hammer for the construction interface.

The opening a specific interface also, naturally, required the closing of all other interfaces that were open; this was done to prevent the overlapping of interfaces and the cluttering of the screen. There was also no need to keep unnecessary interfaces open as they would only take unnecessary resources to render.

The information that needed to be displayed included the player's level, progression towards the next level, progress in their quest, the amount of the three different basic resources they possessed, and the amount of premium resource they possessed. In addition to displaying all this information, it also needed to be updated in real-time as the player progressed in the game.

3.1.2 Resource Exchange Interface

The game also needed functionality for exchanging resources. As mentioned above, the game needed to have support for three different basic resources and a premium resource. The exchanging of resources needed to be done through a dedicated interface, and the final product can be seen in Picture 2.



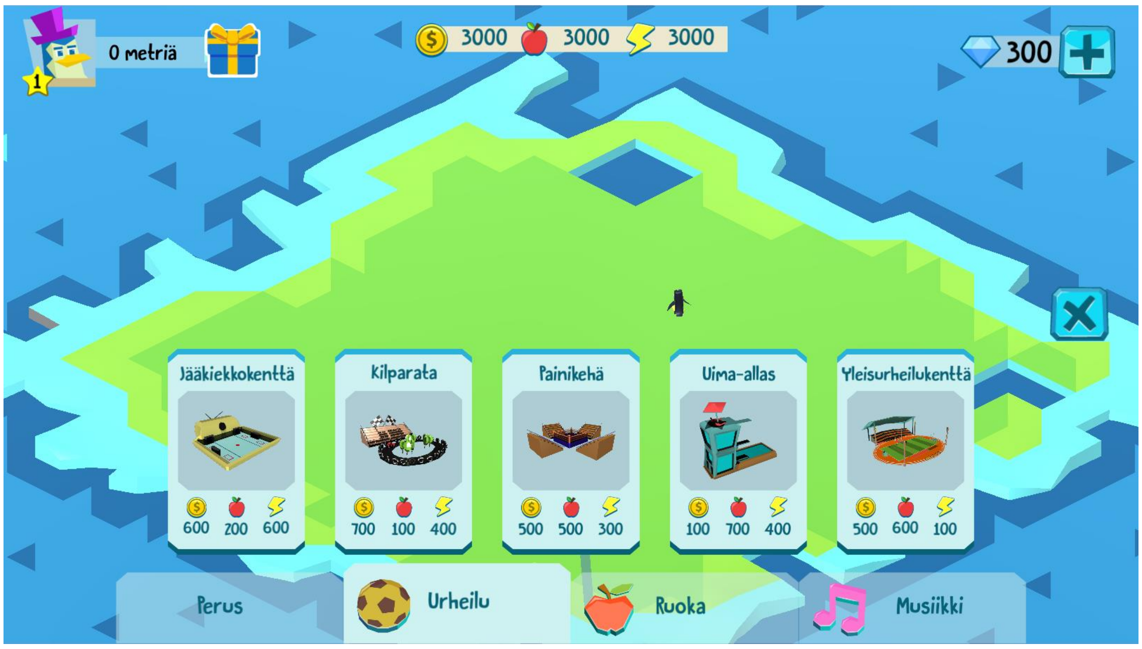
Picture 2. Resource exchange interface for Movenator.

The actual exchange was done in the form of trading in premium resources in exchange for basic resources. The player could choose the amount of resources being traded in and see what they would receive in return.

The interface had functionality for increasing and lowering the amount of premium resources being traded in; this amount was incremented or decremented by 1 per button press. The values on the interface needed to be updated accordingly, and general resource totals needed to be updated whenever the player decided to make the exchange by pressing the corresponding button. The amount of resources possessed by the player was also checked each time an exchange was attempted to prevent any exchanges the player could not afford.

3.1.3 Construction Interface

For city-building, a crucial aspect is constructing buildings. To allow players to do this, a construction interface, seen in Picture 3, was needed; the construction interface would allow the players to choose the buildings they want to construct in their city.



Picture 3. Construction interface for Movenator.

The construction interface needed two different functionalities; these included choosing a genre of buildings to limit the choices, allowing players to find the ones they were looking for more easily, and choosing the actual building they wanted to construct.

The building genres were divided by what the buildings produce; some of the buildings would produce basic resources for further constructions and other gameplay features, while others would produce units for the player to use in other game modes.

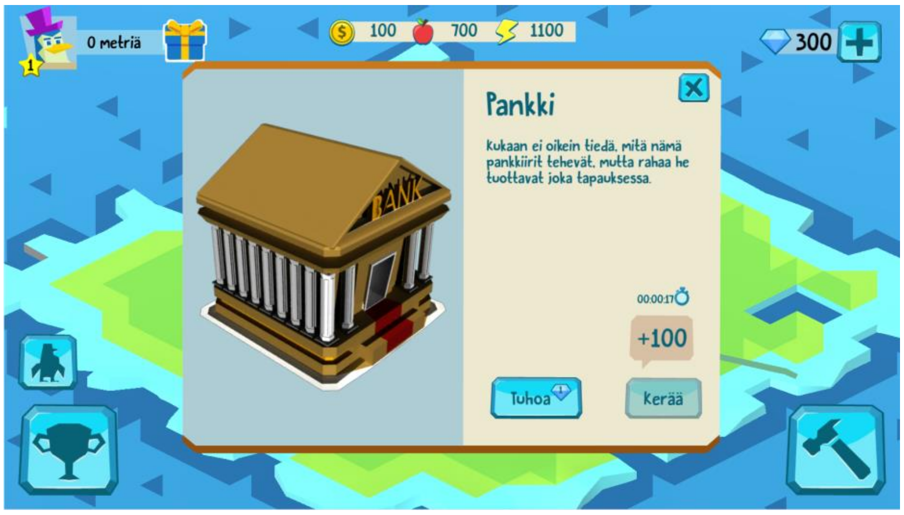
Once the player had chosen the genre of buildings they wanted to see, they needed to be able to browse the buildings in the chosen category. They also needed to see various kinds of information regarding the buildings; this included the name of the building, its image, and the resources needed for its construction. Furthermore, the selection needed to be limited by the available resources so that the player could not choose to construct a building they could not afford.

Additional Unity Editor functionality was also built for the artists working on the project. In Unity, it is possible to build things called prefabs. These prefabs are game objects that consists of multiple different components. These components can range from 3D models to 2D images to scripts of C# or UnityScript code to a variety of different things. The creation of prefabs allows a rapid reproduction of the object, along with other benefits.

As the artists produced buildings that were used in the game, a quick method of creating a new entry for the construction interface was also needed. This involved creating a prefab for a building card for which another person could later attach the building prefab they had created. Then, by pressing a single button, the building card would be filled with all the information that needed to be shown to the player. This also allowed the data to be processed during the development phase, and thus, it reduced the processing time needed when the player started playing the game.

3.1.4 Building Production Interface

Once the player had constructed a building, they needed to be able to examine it and access its features; therefore, the building production interface, seen in Picture 4, needed to be implemented.



Kukaan ei oikein tiedä, mitä nämä pankkiirit tehevät, mutta rahaa he tuottavat joka tapauksessa.

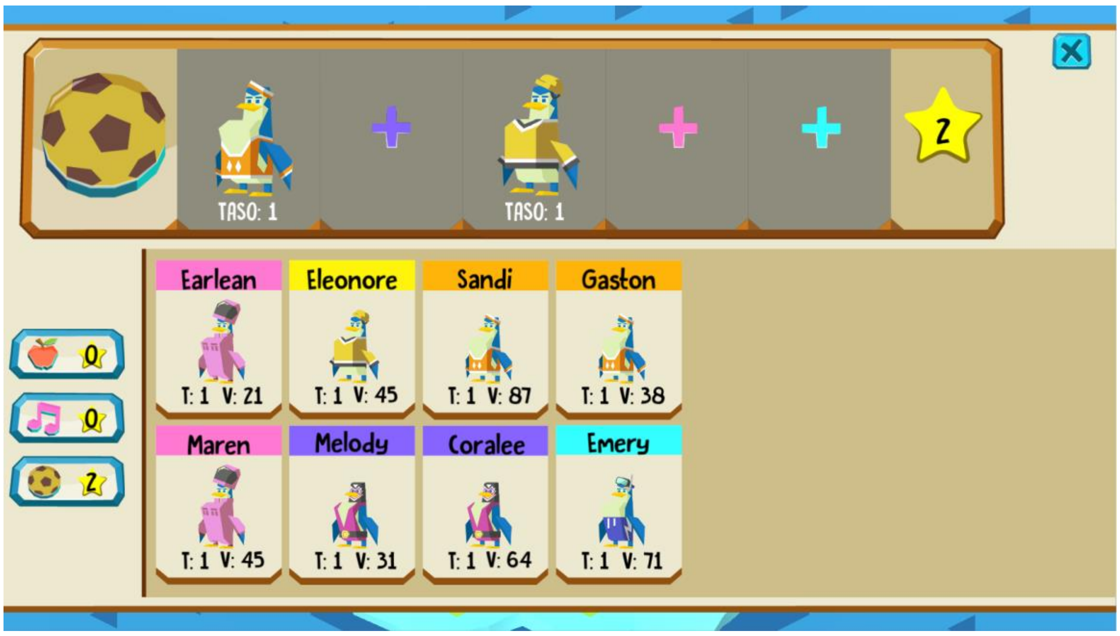
Picture 4. Building production interface for Movenator.

The player needed to be able to see the name, image, and description of the building after opening the interface so they would still remember the building they were examining. They also needed to be able to see the production timer which controlled how much time was left for the player to collect whatever that specific building was to produce; the timer also needed to be updated in real-time, and the player needed to be able to collect the produced items after the timer had reached zero. Additionally, the player needed to be able to destroy the building should they want to do so; this would then reward the player with premium resources.

A separate production interface for each of the buildings would have been unnecessary since the information shown for the buildings was the same. Therefore, a single interface was implemented. This, of course, meant that the data needed to be updated based on the building that player tapped on.

3.1.5 Team Interface

The units produced by certain buildings needed to be presented on a separate interface; the player needed to be able to examine all the units, as seen in Picture 5, they had produced. In addition to examining their units, they also needed to be able to form teams with them. The buildings of the same category would produce units of the same category which could then be used to form teams of that specific category.



Picture 5. Team interface for Movenator.

The team interface was arguably the most complicated one of all the interface that needed to be developed for the Movenator project. It consisted a lot of data that needed to be displayed for the player. In addition to this, the data also needed to be constantly updated as the player progressed in the game and as they acquired more units.

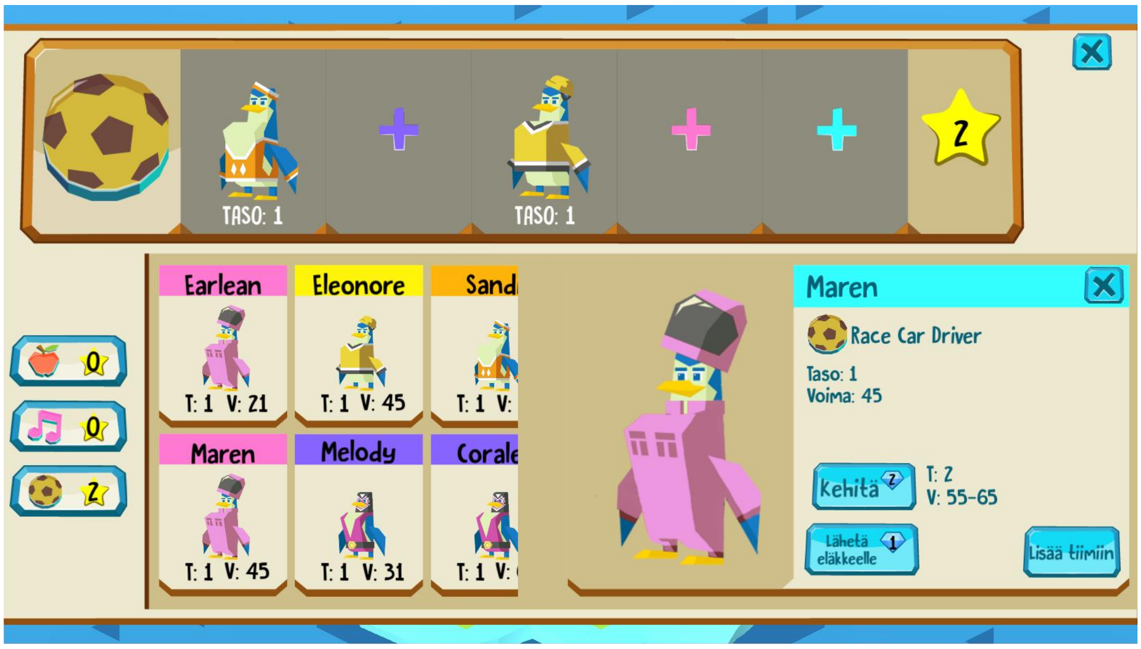
As mentioned previously, every category needed their own team of units. These teams are represented on the left in Picture 5. Furthermore, the level of the team also needed to be displayed and updated as the player progressed. On the top of the interface, the highlighted team needed to be displayed; this team was chosen by tapping on one of the teams on the left-hand side of the interface.

For the highlighted team, several things needed to be displayed; these included a symbol of the team category, all of units on the team and their corresponding levels, and the combined level of the units on the team. If the player had an empty slot in their team of five, a plus sign was displayed for the slot; tapping on that allowed the player to filter the unit cards on the interface to show only the ones that could be used to fill that empty slot.

The bulk of the interface consisted of the unit cards that can be seen in the middle of the screen in Picture 5. Each of the cards were to represent one unit that the player had produced. For the cards, various kinds of information needed to be available for the player to see; this included the name, image, level, and competition power of the unit. In addition to this, the unit card also needed to be highlighted by a specific color. Inside each of the categories, there were 5 different sub-categories of units, represented by the five different colors. Each of the teams contained one slot for each of the sub-categories.

Furthermore, only the relevant unit cards were to be shown for the player; this was done to prevent the player from adding units of incorrect category to a team. The unit cards needed to be filtered every time the player chose to examine a team of a certain category, and they needed to be updated every time any changes had been made to them.

The interface also required further functionality for the players to be able to examine each of the produced units individually; these involved displaying more detailed information and functionality regarding the unit. The unit detail interface can be seen in Picture 6.



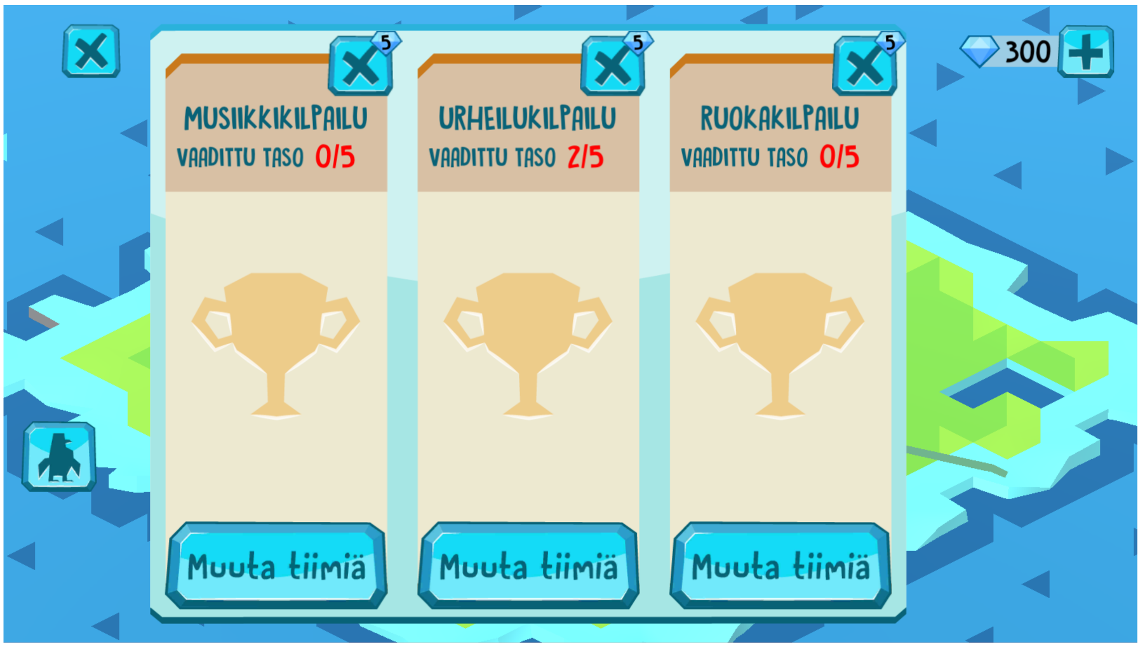
Picture 6. Unit detail interface for Movenator.

The interface was required to display the image, name, main category, sub-category, level, and competition power of the specific unit for the player. Additionally, the player was to be able to upgrade the unit to the next level, with the cost of premium resources, and they needed to see how the unit's competition power would change with the upgrade.

Via this interface, the player also needed to be able to add and remove the unit from the team of its main category; this naturally also required background functionality for tracking the different teams the player had assembled. Furthermore, the player was to be able to delete existing units in exchange for premium resources by "retiring" them, and the amount of the gained resources needed to be tied to the level of the unit; this needed to be achieved because the game would provide the player with new units, and a substantial number of unnecessary units could have ended up clogging up the team interface altogether.

3.1.6 Competition Interface

The last of the interfaces, that were required to be developed, was the competition interface. This interface was set apart from the rest because it is heavily tied to another game mode in the game. Through this interface, seen in Picture 7, the player would be able to join a minigame with a team of their choosing, and the chosen team would then affect the gameplay.



Picture 7. Competition interface for Movenator.

The interface needed to display 3 different minigames to the player; the player could then choose one from these that they could then play. All the minigames required that the player had a team of the same category as the minigame itself; another requirement was that the player's team needed to be of high-enough level before they could access the minigame.

The interface needed further functionality in the form of so-called "re-rolling" of the minigames; in practice, the player could spend a specific amount of their premium resources to discard one of the minigames, and the game would then generate one from a different category for them to play.

Each of the competition elements also needed a button to act upon the competition. If the team level was not high enough to enter the competition, the player was taken to the team interface where they could manage the team in question; and if the level requirement was met by the player's team, the player was able to enter the competition.

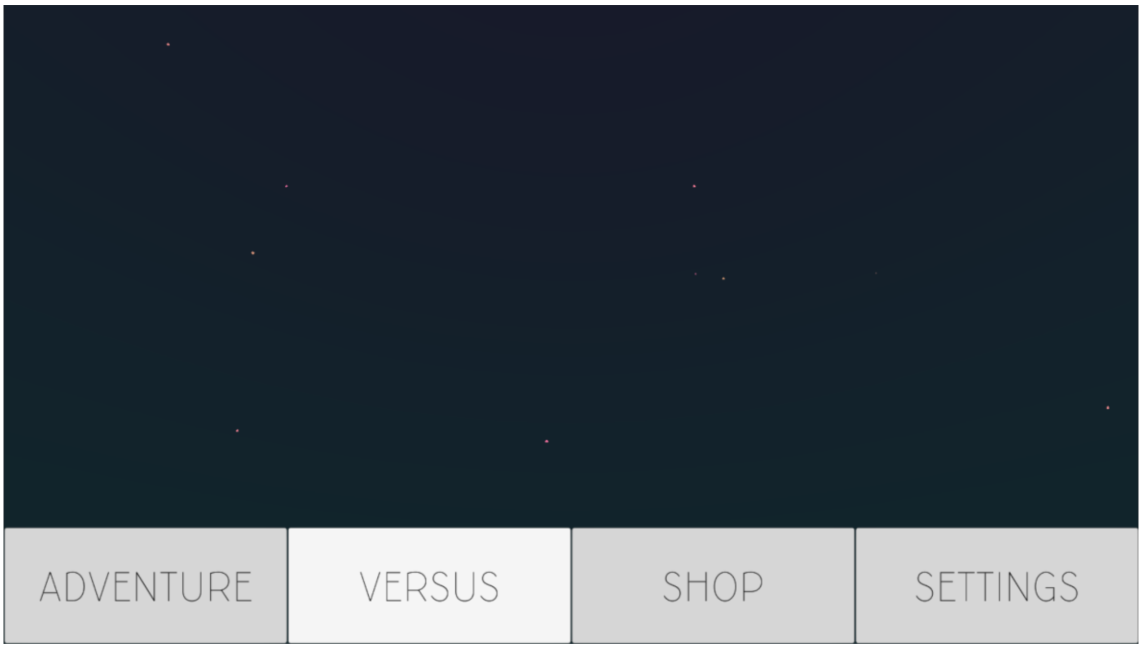
3.2 Mini Golf Universe

Arguably, for the Movenator project, there was more time to focus on the development of the UI which allowed multiple iterations. Thus, the final products for the prototype that was used for user testing was something that also could have been perceived to be much further along in development.

Technically, this was true as well. However, the requirements were quite different as well. For the Movenator project, a lot of UI graphics were implemented as well. However, for Mini Golf Universe, it was essential to build the functionality for the first prototype so that it could be tested and further iterated. For the UI of Mini Golf Universe, the implementation of actual graphics was also of no importance.

3.2.1 General Menu Interface

The purpose of the main menu in Mini Golf Universe was to access different sub-menus including the Adventure, Versus, Shop, and Settings menus. The general menu controls can be seen in Picture 8; these four buttons shown cover most of the general menu interface functionality.

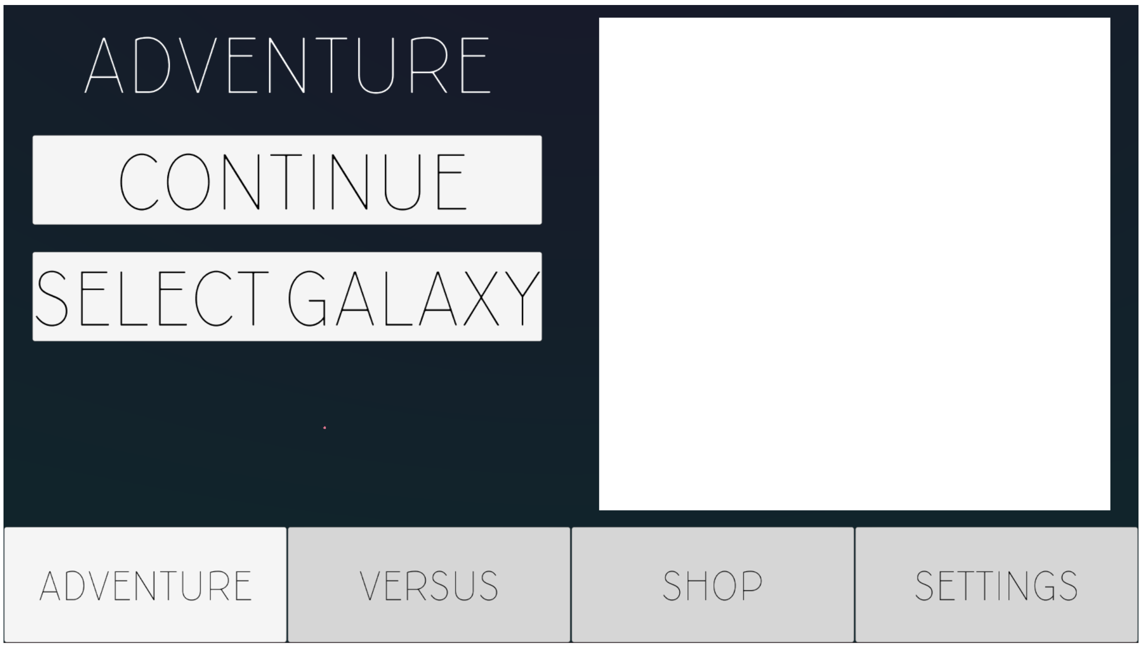


Picture 8. General menu interface for Mini Golf Universe.

These buttons are used to access the menus. Each of the buttons had specific content attached to them. When one of the buttons was tapped, it triggered an event that was then handled by all the buttons; this caused the buttons to individually show or hide their content based on which of the buttons was pressed.

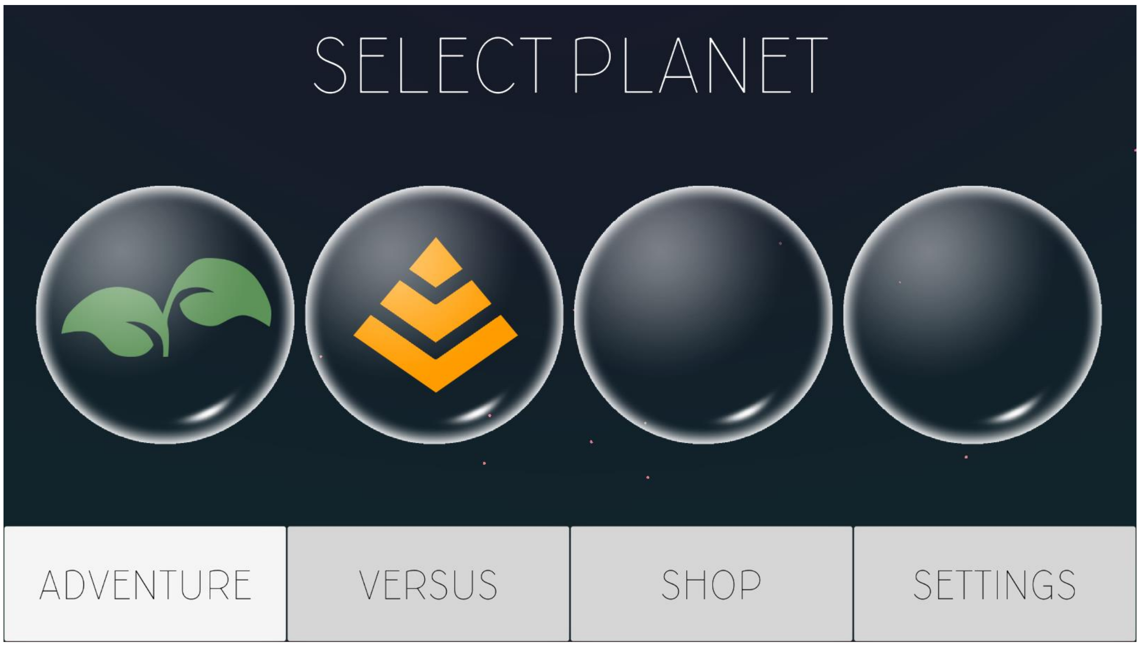
3.2.2 Adventure Menu

The adventure menu was built to allow the player to access the single player content of the game. The adventure menu contained three different views inside it: the default view, the planet view, and the galaxy view. The default view is shown in Picture 9.



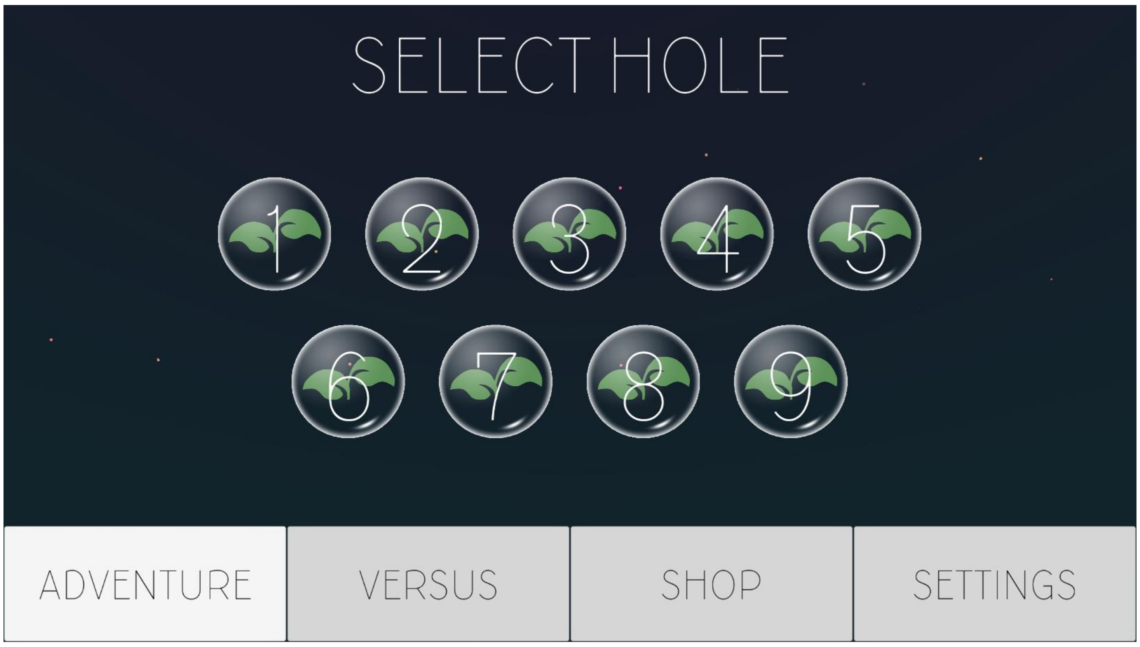
Picture 9. Default view of the adventure menu for Mini Golf Universe.

The purpose of the default view was to allow the player either to quickly continue playing the set of levels they had previously been playing, or to switch to another set of levels. The default view was also to show an image reflecting the set of levels the player had previously been playing. If the player had not yet started playing any set of levels, they would instead be shown the level set selection view, titled “galaxy view”, seen in Picture 10.



Picture 10. Galaxy view of adventure menu for Mini Golf Universe.

As the game was to feature multiple sets of levels, the ”galaxy view” allowed them to choose a set of levels they wanted to play. By tapping on any of the sets shown, the player could then proceed to choosing the level they wanted to play; this was done in the “planet view” shown in Picture 11.

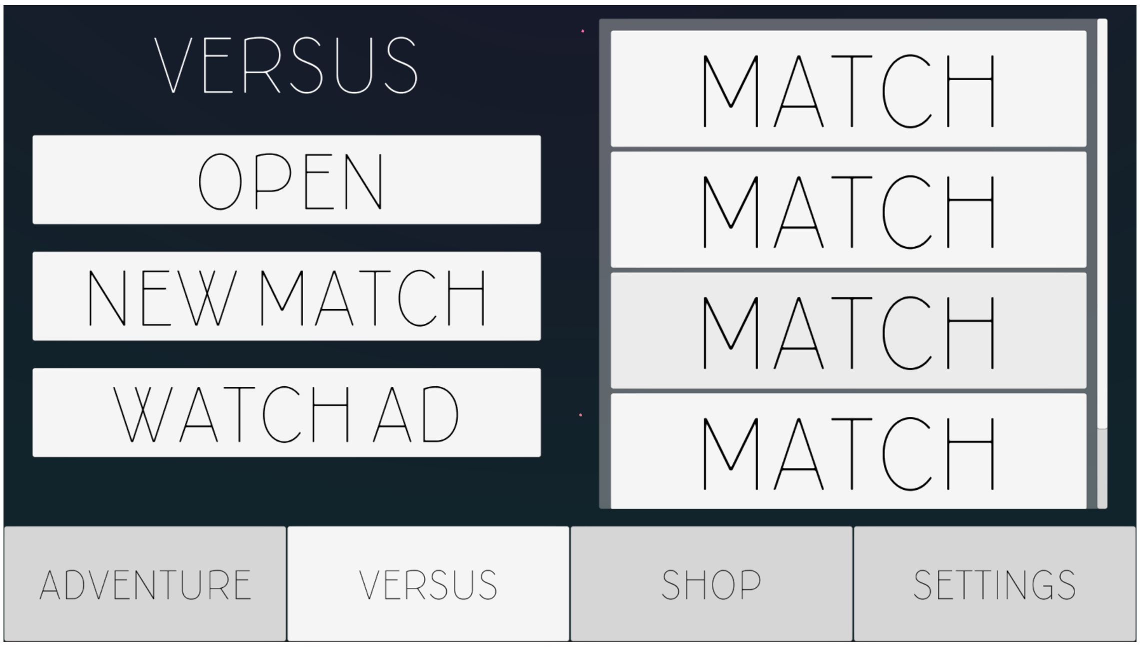


Picture 11. Planet view of adventure menu for Mini Golf Universe.

The ”planet view” allowed the player to choose a level to play from the set of levels they had previously chosen. By tapping on any of the level icons, they would then be launched into that level. Later functionality would also include level statistics shown for the player, but this was not yet included in this prototype.

3.2.3 Versus Menu

The versus menu was implemented to allow the player access to the multiplayer components of the game. Additionally, the player would be able to open prizes for playing the game and watch video ads to gain in-game resources. The versus menu can be seen in Picture 12.



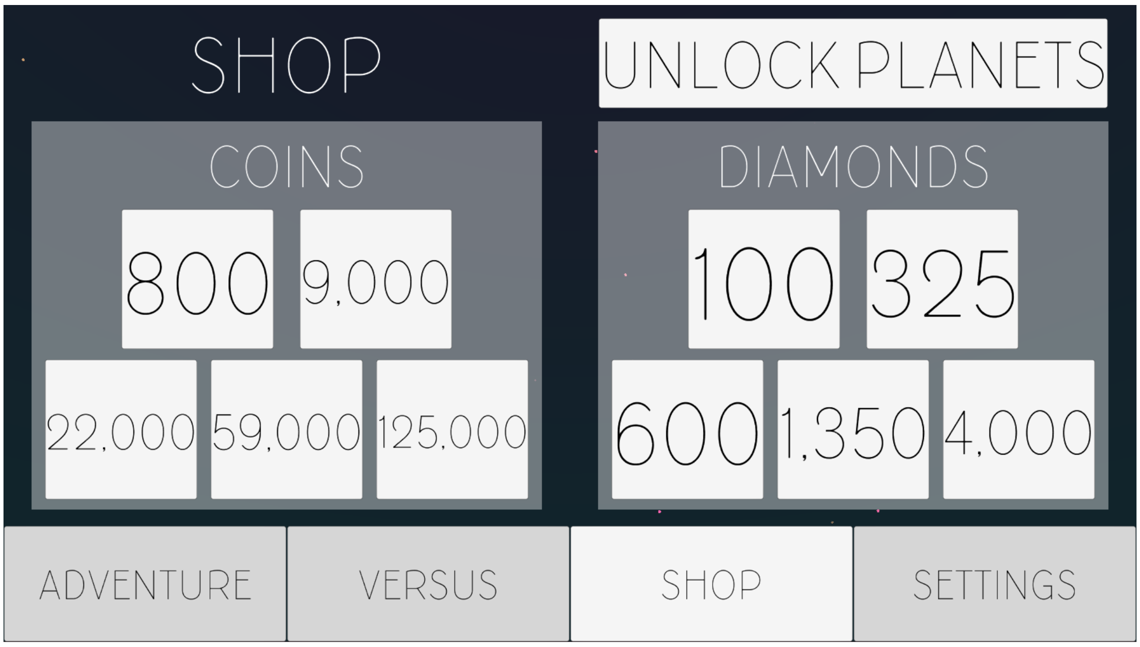
Picture 12. Versus menu for Mini Golf Universe.

The right half of the screen was reserved for on-going online matches that the player was taking part in. This was implemented ahead of the multiplayer functionality, but there would be a button representing each of the matches in a scrollable list, allowing any number of on-going matches. By tapping on any of them, provided that it was the player’s turn, they could continue playing the match.

The buttons on the left-side of the screen allowed three kinds of functionality. The player could press a button a few times a day to allow them to open a prize that would reward them in-game currency. They could press another button to start a new match, which would then add a button to the list of on-going matches. They could press a third button to watch a video ad to gain additional in-game currency.

3.2.4 Shop Menu

The shop menu was built to allow the player to purchase several types of currencies as well as additional sets of levels. The shop menu was also to feature additional possibilities in future versions; the menu is shown in Picture 13.



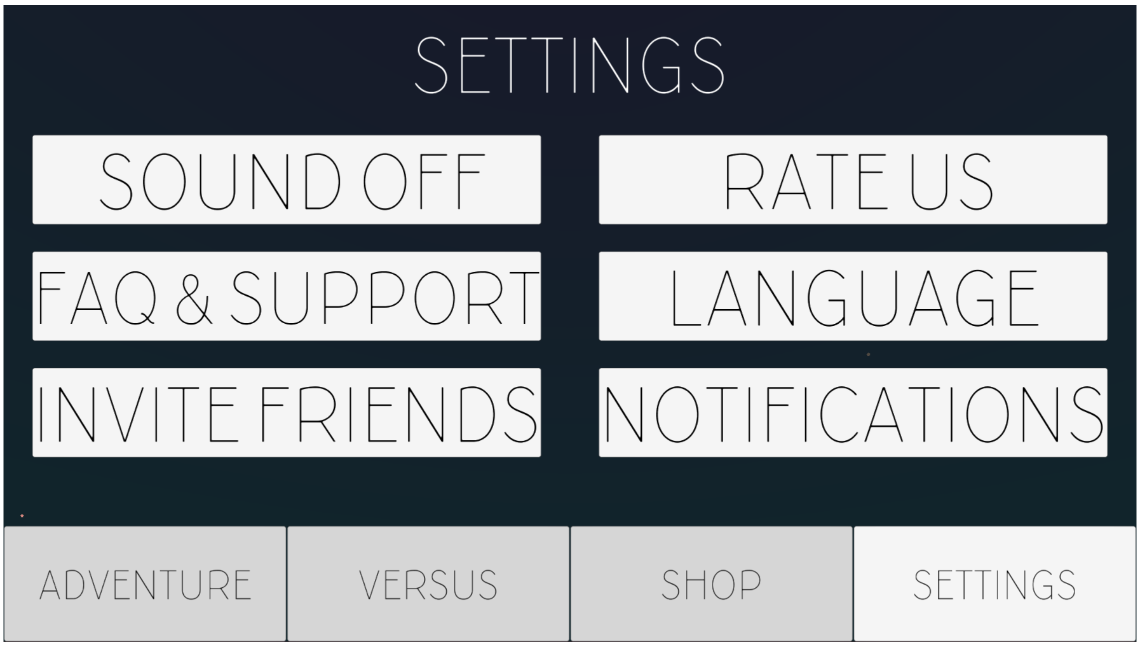
Picture 13. Shop menu for Mini Golf Universe.

The menu would allow the player to make three kinds of purchases: level sets for coins, coins for diamonds, and diamonds for real money. By tapping on a purchase option, an online confirmation would be made to validate the purchase. Future additions included several types of customization options that the player could purchase. This would then, naturally, cause additional changes to the layout.

3.2.5 Settings Menu

The purpose of the settings menu, seen in Picture 14, was to gather all the different settings, options, and miscellaneous features that the player could access into one place.

Many of the functionalities had not yet been implemented for the first menu prototype, and many of the functionalities also included connecting to services outside of the game.



Picture 14. Settings menu for Mini Golf Universe.

The menu consisted of simple buttons. The planned functionality for these buttons included turning the game audio on and off, accessing the support channels of the game, inviting friends to the game, rating the game on Google Play or App Store, changing the interface language, and modifying the notification settings.

4 TECHNICAL IMPLEMENTATIONS

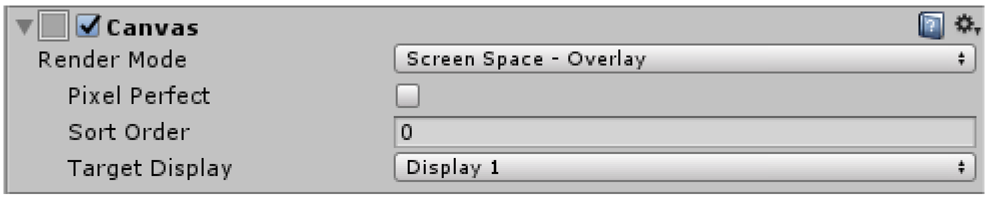
This section will focus on providing information on how the final implementations were done and what UI and optimization tools were used in the process. This offers insight as to why certain decisions were made. As the process is presented, the different UI tools and their fundamental functionalities are introduced. Testing of the UIs was conducted on a low-end mobile device to offer best insight into optimization; the device in question was an LG Spirit 4G LTE H400N.

4.1 Movenator

Movenator was a large project, and the UI duties involved developing a rather complex UI with a lot of different kind of functionality. On top of this, it should be noted that the functionality for the Movenator project was focused on providing support to the gameplay functionalities. When comparing the two projects discussed in this thesis, the Movenator project is clearly the larger one.

4.1.1 User Interface in General

The Unity game engine features a pre-made Canvas component that can be attached to a Game Object to create a Canvas object. The Canvas object is something that holds all the different UI elements that are shown for the player, and the Canvas component features different options to alter the way the UI elements behave. The Canvas component and its options can be seen in Picture 15. (Unity Technologies 2017h.)


Canvas
Render Mode: Screen Space - Overlay
Pixel Perfect: ☐
Sort Order: 0
Target Display: Display 1

Picture 15. Canvas component.

The most important option at this point is the Render Mode. The Render Mode has three different options: “Screen Space – Overlay”, “Screen Space – Camera”, and “World

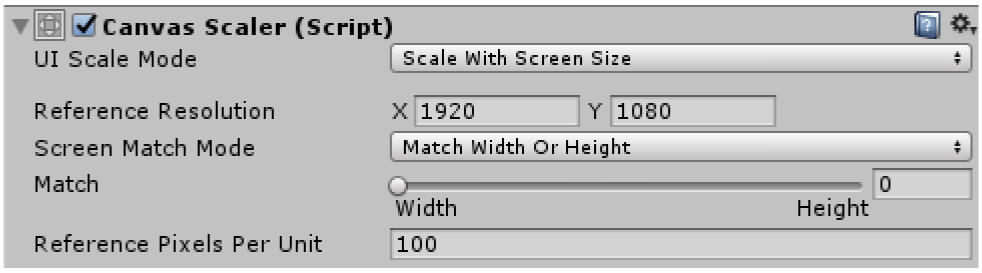
Space". The different options affect how the Canvas is rendered, i.e. how the player will see it on their screen.

The "Screen Space – Overlay" mode is the one we wanted for the implementation of the UI. In this mode, all the UI elements are rendered on top of all the objects in the game world (Unity Technologies 2017h). This way, the UI elements are always visible for the player and no game object will ever block any of them.

Alternatively, the "Screen Space – Camera" mode could be used to render the UI elements with a specific Camera in the game world, and they could be set within a certain distance of that Camera. In this mode, the settings of the Camera object would also affect how the UI would be rendered for the player. This could, for example, allow the addition of perspective for the UI, should that be desired by the developer. (Unity Technologies 2017h.)

Lastly, the "World Space" mode is very different from the other two modes. When the Canvas is set to "World Space" mode, it can be placed anywhere in the game world, rather than being tied to always appear on the screen (Unity Technologies 2017h). This has its own purposes, but for the sake of implementing most of the interface elements in the Movenator project, the "World Space" mode was of no use.

Another important part of the Canvas object is the Canvas Scaler script component; it is also a pre-made part of the Unity game engine. The Canvas Scaler script component and its options can be seen in Picture 16.


Canvas Scaler (Script)
UI Scale Mode: Scale With Screen Size
Reference Resolution: X 1920 Y 1080
Screen Match Mode: Match Width Or Height
Match: 0 (Width – Height)
Reference Pixels Per Unit: 100

Picture 16. Canvas Scaler component.

The Canvas Scaler script component also has two specific options that should be focused on the most. First of them is the UI Scale Mode. The UI Scale Mode "determines how UI elements in the Canvas are scaled". It features three different options: "Constant Pixel Size", "Scale With Screen Size", and "Constant Physical Size". The names are descriptive so it is easy to deduce what their functions are. For the Movenator project, the UI elements needed to scale along with the game world, and that is why the "Scale With Screen Size" was chosen to be used. (Unity Technologies 2017i.)

Additionally, attention should be paid to the Reference Resolution. This is the resolution for which the UI is designed; it is also used as a reference when the player has a screen with a different resolution and the UI needs to be scaled. It was set to 1920 by 1080 as this is an extremely common resolution. When later adjusting the positions and sizes of the UI elements, that is also done with this resolution in mind.

For many of the game's interfaces, the initial idea was to have them slide into the game view, making them look better for the player. However, this was not yet implemented for the current prototype due to time constraints. These sliding elements included the resource exchange interface, construction interface, building production interface, team interface, and competition interface.

Even though the slide functionality could not be implemented, the elements' effect on the performance was still measured, should they have been active in the game; this was done because "active UI elements are still calculated and added to the draw calls even though they aren't seen". (Tantzy Games 2016).

The effect was measured by examining the number of draw calls made by using the Profiler tool. To give context to the number of draw calls, their number was measured with only the general game interface open; in this situation, the total number of draw calls amounted to 32.

Before measuring the number of draw calls again, the interfaces were activated and moved outside of the Camera's view. This meant that they could not be seen, and the screen for the player seemed exactly like before; nevertheless, the number of draw calls now amounted to 78. This is more than double the amount of draw calls that were made when those interfaces were not active; that is certainly a huge increase with no visible gain.

Deactivating an object in the game means, of course, that none of its functionality will work, and none of the attached scripts are run. This can problematic if the UI elements contain functionality that involves data being processed even when the element in question is not shown on the screen. Fortunately, there was a way to both keep the UI elements active and not increase the number of draw calls.

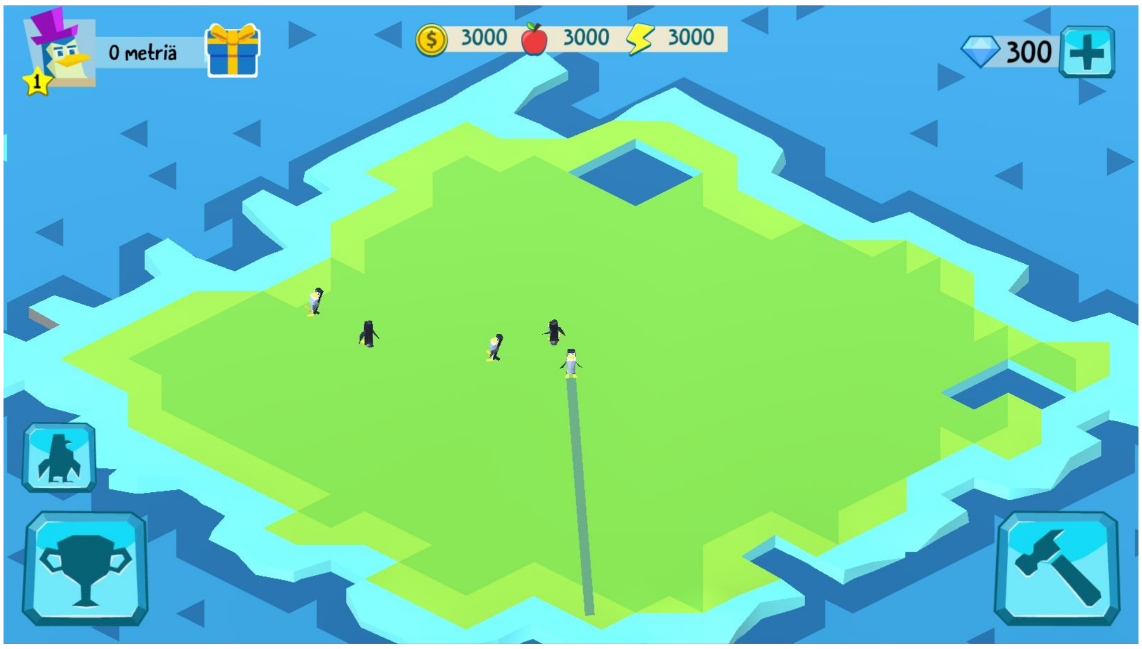
The Unity game engine features a Rect Mask 2D component. This component can be used as a part of the Canvas to hide the UI elements that the player cannot see (Tantzy Games 2016). The component thus provides better performance for the user, but in development, it should be noted that, as the name suggests again, this “only works in 2D space” (Unity Technologies 2017j). Before measuring the number of draw calls again, a Rect Mask 2D component was attached to the Canvas. The number of draw calls was now measured to be 32; the same number of draw calls made in the beginning while none of the specified UI elements were active.

As was mentioned before, the slide functionality was not implemented for this prototype of the project. Thus, the method of activating and deactivating the interfaces was used to reduce the amount of draw calls made.

For the Movenator project, the default Text component of Unity game engine was used to implement all the text elements. However, Unity Technologies acquired the TextMesh Pro asset midway the development process of the Movenator project. The new component should replace the old Text components, and it should offer optimized performance for mobile platforms as well. The current Unity Text component is known to have certain performance problems. However, the change was not deemed necessary for the sake of this prototype. (Tantzy Games 2016; Winter 2017.)

4.1.2 General Game Interface

To control the interactions between the various interfaces of the game, a UIManager script was written. One of its functionalities was to keep the resource totals, shown in the top of the screen in Picture 17, up-to-date as the game progressed; this required interaction between the UIManager and another script, ResourceManager.

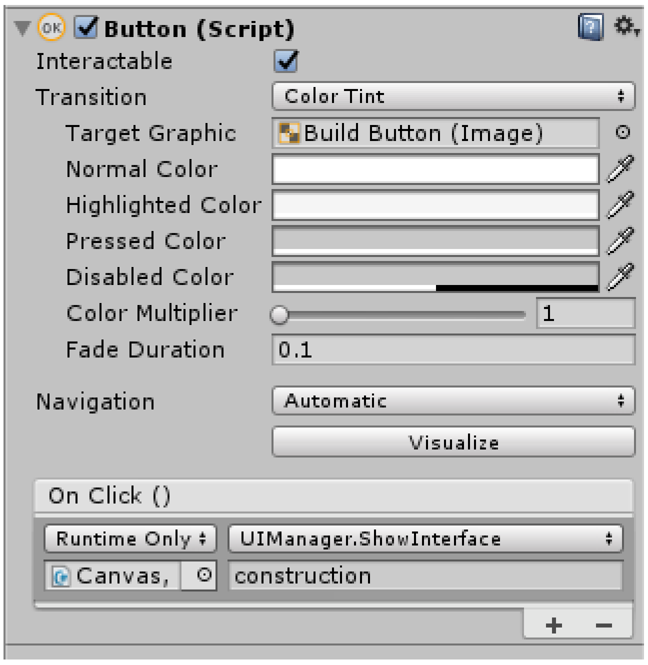


Picture 17. Elements of general game interface for Movenator.

ResourceManager was written to keep track of the resources the player possessed. Its main functionalities also included methods for adding, removing, and checking resources; furthermore, it featured an event for whenever the resource totals were changed.

The UIManager had its own method for updating the resource totals shown. This method was set to subscribe to the event contained in an instance of the ResourceManager. Whenever the resource totals were changed, the event was triggered, and the UIManager could react upon it by updating the resource totals shown for the player.

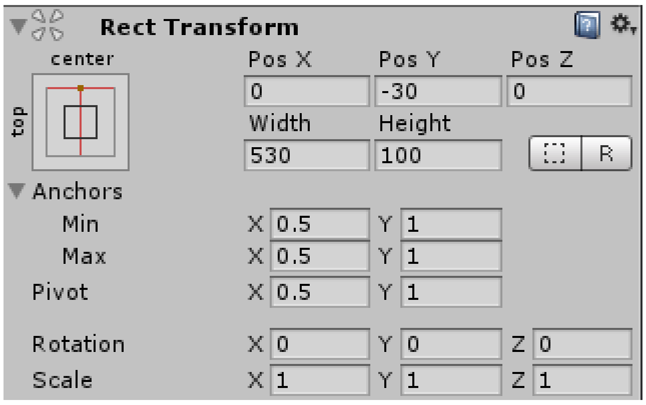
Another functionality for the general game interface was to open the other interfaces that were included in the game. For this, another functionality of the UIManager was developed; this was a method that could be called on the UIManager instance which allowed to change the interface shown by passing a string identifier as a parameter of the method. This way, the method could be used both in other scripts and directly with the Button script components that were part of the Button objects in the general game interface; an example of the Button script components can be seen in Picture 18.


Button (Script)
Interactable ✓
Transition: Color Tint
Target Graphic: Build Button (Image)
Normal Color
Highlighted Color
Pressed Color
Disabled Color
Color Multiplier: 1
Fade Duration: 0.1
Navigation: Automatic
Visualize
On Click ()
Runtime Only
UIManager.ShowInterface
Canvas,
construction

Picture 18. Button component.

The Button script component features many functionalities. At this point, however, the focus is on the “On Click ()” container and its content. “On Click ()” refers to an event “that is invoked when when a user clicks the button and releases it”. (Unity Technologies 2017k). With the Button script component, the developer can directly set what should happen when this Button is pressed. First, a reference to the game object which holds the desired UIManager script is set for the field in the bottom-left; in this case, the UIManager script had been attached to the Canvas object. Secondly, the desired script and its method was chosen from the dropdown menu in the top-right. Finally, the parameter for the method was set in the text field in the bottom-right. This allowed the “On Click” event to directly call the method in the UIManager instance.

All the UI elements in Unity feature a Rect Transform component. This component determines affects the position, rotation, and size of the element; an example of a Rect Transform component can be seen in Picture 19.


Rect Transform

center	Pos X	Pos Y	Pos Z
top	0	-30	0
	Width	Height	
	530	100	R

Anchors

Min	X 0.5	Y 1	
Max	X 0.5	Y 1	
Pivot	X 0.5	Y 1	
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Picture 19. Rect Transform component determining the position of object.

This example is of the element holding the basic resource totals. As seen in the picture of the general game interface, this element was positioned in the top-center of the screen. This was achieved by setting the Anchors of the element there. The Anchor values, set between 0 and 1, control where the element is anchored on its parent element, which in this case was the Canvas itself as the element in question was a direct child object of the Canvas.

When both the X and the Y values are set to 0, the element is anchored to the bottom-left corner of the parent element; when they are both set to 1, the element Is anchored to the top-right corner of the parent element. Setting both the minimum and maximum values of an Anchor to the same value allows the size on that axis to be determined manually, as was done with both the width and the height of this element.

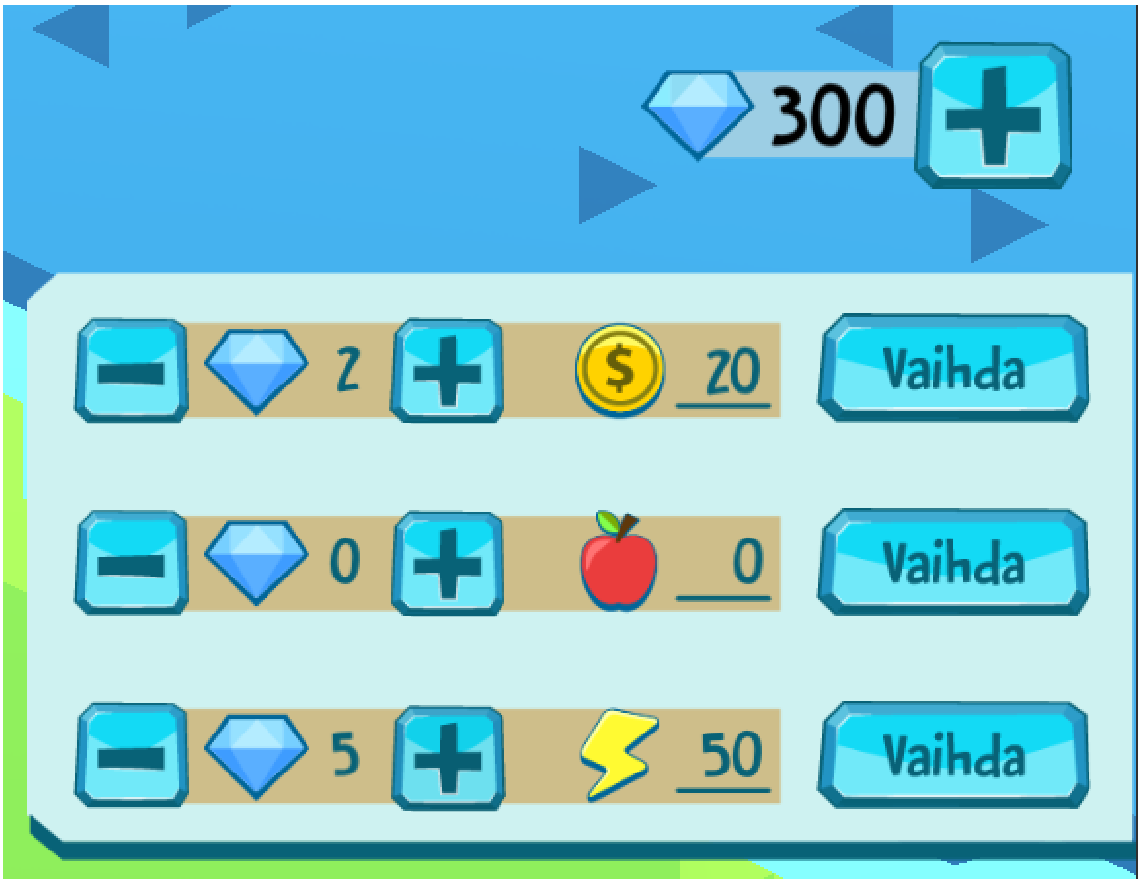
The Pivot was set to 0.5 on the X axis, which is the horizontal center of the element; on the Y axis, it was set to 1 which is the top of the element on the vertical axis. This way, the Pivot was set to the top-center of the element.

After setting both the Anchor and the Pivot values, the objects position can be further adjusted by adjusting the Position values. For these elements, only the X and Y values were of importance since the UI elements were always going to be rendered on top of the other game objects, as was discussed before. Setting the Pos Y value to -30, positioned the elements Pivot 30 pixels below the Anchors; in this case, the top-center of the resource totals element was 30 pixels below the top-center of the screen.

This method of anchoring the element on the screen was used for all the UI elements in the general game interface. This is because the purpose for all of them was to appear on the edges of the screen, and this method allows that on all resolutions.

4.1.3 Resource Exchange Interface

The resource exchange interface included functionality for exchanging premium resources for basic resources; it was one of the simplest interfaces implemented for the game. This interface, seen in Picture 20, consisted of three similar rows; each of the rows was for one of the basic resources. Inside the row, there were buttons for both decreasing and increasing the amount of premium resources being traded in. Next to it, the amount of basic resources, that would be gained by the exchange, was shown. These values updated with each button press. This process was controlled by a custom ResourceExchange script component.

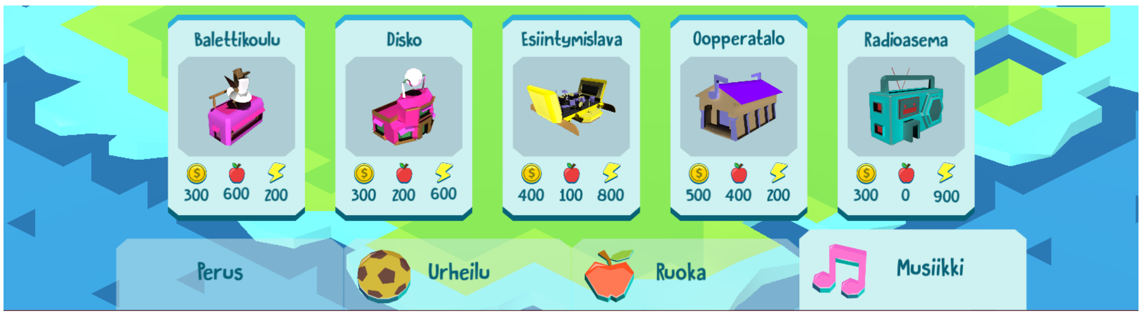


Picture 20. Elements of resource exchange interface for Movenator.

When the player decided to make the trade by pressing on a button on the right, the “On Click” event would call on the ResourceManager instance to check if the player could afford the trade. If so, it would also call the methods for adding and removing reduces accordingly, which would then trigger the event mentioned before and update the resource totals shown on the screen.

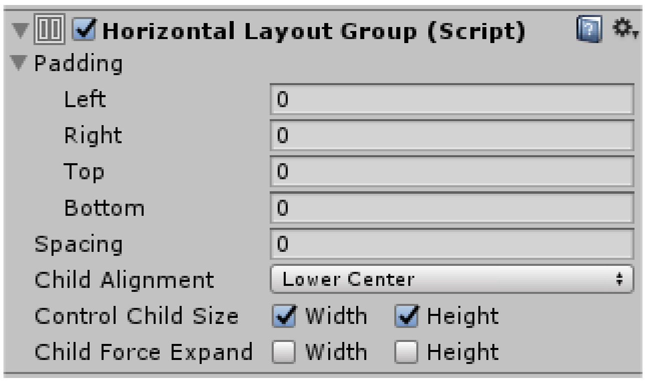
4.1.4 Construction Interface

The construction interface, as seen in Picture 21, featured the functionality for choosing the building that the player wanted to construct. As it was discussed before, the buildings in the game were divided into different main categories based on what they produce.



Picture 21. Elements of construction interface for Movenator.

These category selection buttons in the bottom were set as a child object of an element with a Horizontal Layout Group script component attached to it. This component allows easy positioning of UI elements that are part of a unified group; an example of the script component can be seen in Picture 22.


Horizontal Layout Group (Script)

Padding		
Left	0	
Right	0	
Top	0	
Bottom	0	
Spacing	0	
Child Alignment	Lower Center	
Control Child Size	✓ Width	✓ Height
Child Force Expand	☐ Width	☐ Height

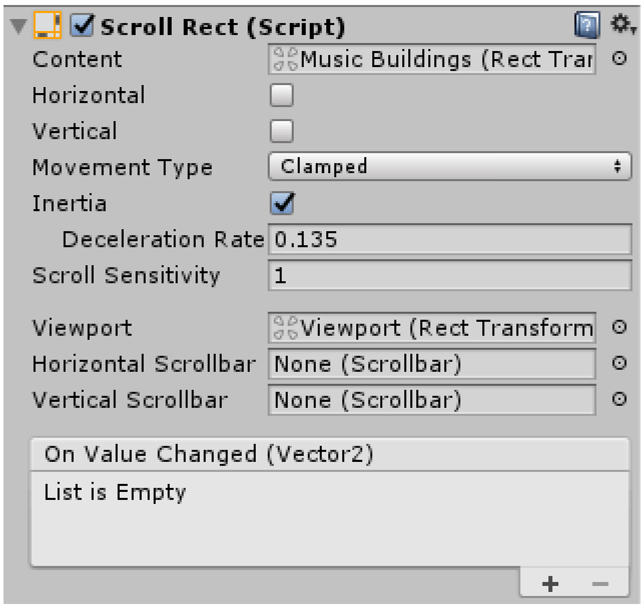
Picture 22. Horizontal Layout Group component.

As can be seen in Picture 21, the buttons are positioned right next to each other. This was intended behavior, and it is caused by the Spacing value of 0. The parent object, on which this component was attached, horizontally covers the whole screen. Therefore, setting the Child Alignment is important. Basically, it sets the pivot point of the objects in the Horizontal Layout Group. As it is set to Lower Center, should there be only one child added to the group, it would be in the bottom-center of the parent element.

Each of the buttons added to the Layout Group has its own size. Setting both the Width and Height values to true on the Control Child Size option gives the script component the authority to resize the elements so that they all fit on the area of the parent element, should the child elements in their original size take more space than that. This is useful because it allows to constrain the elements to the area that was set for the parent element.

It should be noted that the use of Layout Groups can be expensive performance-wise. This is because they “are re-evaluated every time they are marked as dirty, which basically means lots of calculations any time anything is changed”. Another solution would be to use the Anchors of the Rect Transforms, in a separate way than the one introduced before, to achieve the same effect. This is, of course, manual work, and because the functionality for adding new buttons dynamically was needed, the implementation was done using a Layout Group. (Tantzy Games 2016.)

The category buttons’ Button components’ “On Click” events contained a reference to the ConstructionInterface script and its method for changing the content. In this context, the content refers to the content of the Scroll Rect script component on the Scroll View object; an example for this can be seen in Picture 23.

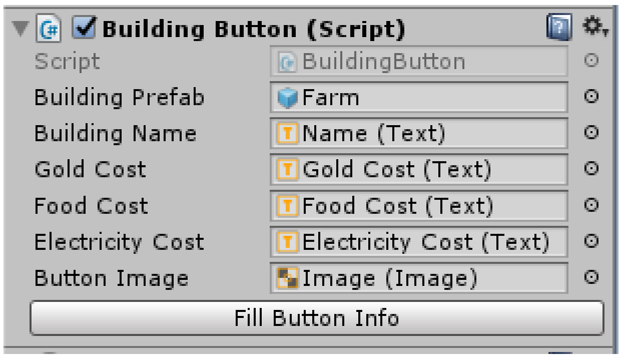

Scroll Rect (Script)
Content: Music Buildings (Rect Trar
Horizontal: ☐
Vertical: ☐
Movement Type: Clamped
Inertia: ✓
Deceleration Rate: 0.135
Scroll Sensitivity: 1
Viewport: Viewport (Rect Transform
Horizontal Scrollbar: None (Scrollbar)
Vertical Scrollbar: None (Scrollbar)
On Value Changed (Vector2)
List is Empty

Picture 23. Scroll Rect component.

The Scroll View object is a pre-made Unity object. Using a Scroll View object allows the developer to quickly build a UI element which has scrollable content. This content can be almost anything in the UI; it could be plain text, a set of images, or even a set of button prefabs, as is the case with the construction interface.

The Content field of the Scroll Rect component requires a reference to the Rect Transform of the object that is to be scrolled. This is the value that the category buttons would change. Each of the various categories featured their own Game Object with a Rect Transform. These objects were placed on top of each other, at the same position on the screen. As the Content was changed, a method was called on the ConstructionInterface script. This method did three things: it activated the Game Object featuring content of the corresponding button’s category, it deactivated the Game Objects for other categories, and it set the activated Game Object’s Rect Transform component as the Content for the Scroll Rect component, allowing the content to be scrolled.

For each of these objects that contained the button prefabs, a Horizontal Layout Group component was also attached. This was done for the same reason as before; there was need to allow the quick addition of new buttons in the future without having to manually adjust the Anchor values of the buttons’ Rect Transforms. Each of the building buttons featured a custom BuildingButton script component, seen in Picture 24.



Building Button (Script)

Script	BuildingButton
Building Prefab	Farm
Building Name	Name (Text)
Gold Cost	Gold Cost (Text)
Food Cost	Food Cost (Text)
Electricity Cost	Electricity Cost (Text)
Button Image	Image (Image)

Picture 24. Custom Building Button component.

This component was part of the functionality for spawning buildings in the game world, but it also featured functionality for the development of the UI. As was mentioned before, the artists on the project developed buildings and created so-called prefabs of them. The game then needed a button in the construction menu for all of them. Therefore, the BuildingButton component featured the Building Prefab field.

The artist could simply add an empty button to the Layout Group of a category, add their prefab reference to its BuildingButton component, and click on the Fill Button Info button on the component. This custom button would then take all the information from the prefab the artist had created and place it on the button. This way, the artist did not need to set the building name, image, and cost for each of the buttons.

Another upside for the implemented functionality was performance. Because all the information for the buttons was there when the game launched, there was less processing needed on startup.

However, an issue with the performance was noticed during tests for which both the Profiler and the custom FPS counter was used. The FPS counter was used to check the overall performance and the Profiler was used to determine where the problem was in case there was one.

While the game could regularly be run at 60 FPS, which is considered a very acceptable framerate on any device, on a low-end smartphone, the framerate dropped down to approximately 45 FPS immediately upon opening the construction interface.

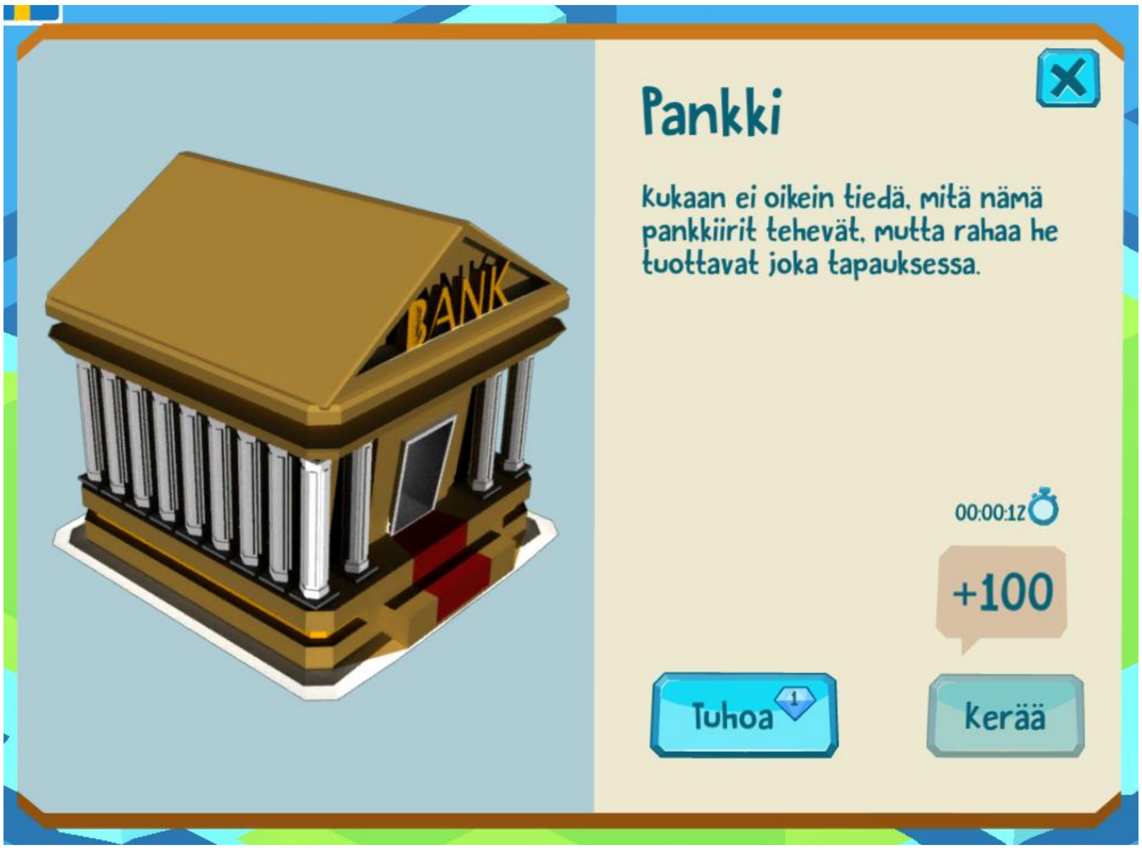
As it climbed back to 60 FPS upon closing the interface, it was easy to determine that this specific interface was the reason for poor performance. However, there was still the need to determine which part of the interface was causing problems. Upon using the Profiler, it could be noticed that the ScrollRect script was using a lot of resources.

Even though the pre-made Scroll View object is easy to use, it is also known for causing performance issues (Tantzy Games 2016); however, this still needed verification. By deactivating the Scroll View object, the framerate rose back up to 60 FPS. The Scroll View was still something that was needed so the problem needed a solution. A known improvement is to replace the UI Mask component, which comes pre-packaged with the Scroll View object, with a Rect Mask 2D component, which was also used with the Canvas object.

Once the improvements had been made, the performance was tested again. This time, the FPS counter showed a solid 60 FPS, even after opening the interface. The performance was deemed to be good and the problem solved.

4.1.5 Building Production Interface

The building production interface was something that needed to appear whenever the player tapped on one of their buildings; an example can be seen in Picture 25. This allowed the player to examine the building information as well as use functionalities related to that building.


Pankki
Kukaan ei oikein tiedä, mitä nämä pankkiirit tehevät, mutta rahaa he tuottavat joka tapauksessa.
BANK
00:00:12
+100
Tuhoa
1
kerää

Picture 25. Elements of building production interface for Movenator.

Whenever a building was clicked and the interface was open, the information regarding the building was filled in the interface; this involved the building image, name, description, production timer, and the amount of resources they would produce. This was read directly from the building's own Building script component which contained this information.

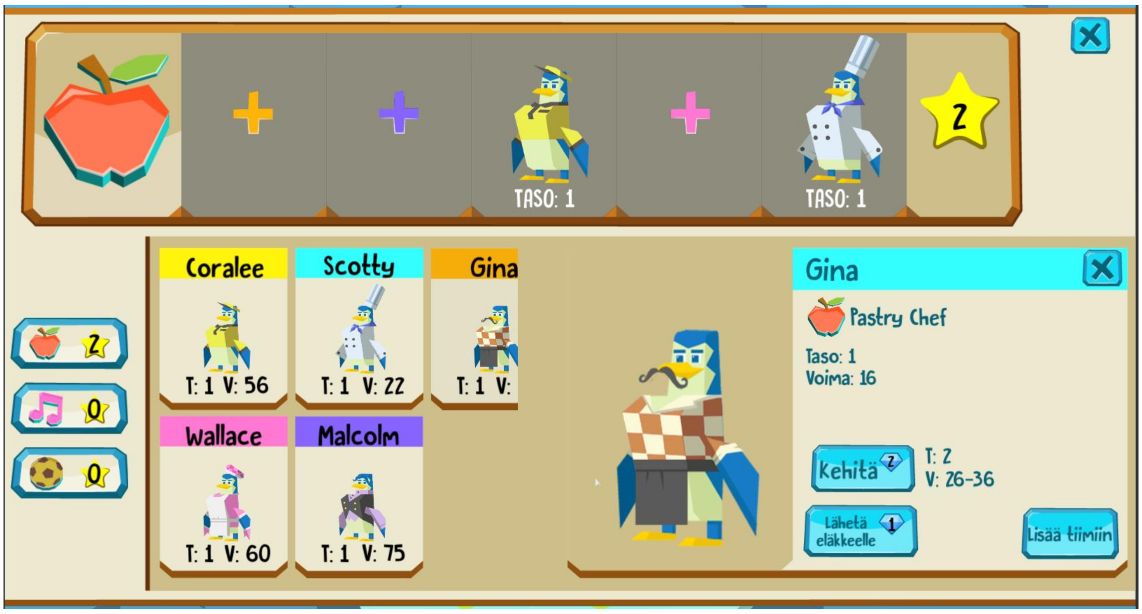
Most of this was simple, as it involved only reading data from another script, but the tracking of the production timer was more complicated. Each of the building's scripts had their own timer which started running when the building's construction phase was finished. The script then featured an event which was triggered every second and passed on the current time remaining in the building's production phase.

The building production interface did not have a script of its own, but its functionality was handled from the UIManager script instead. Whenever the interface was opened, the UIManager would add its listener method to the corresponding building's Building script's event. This way, the correct production timer was always tracked. Whenever the interface was closed, the listener was removed from the previous Building script's event to avoid conflicting data for the interface.

The interface also had buttons for destroying the building and collecting the products of the building. Destroying the building gave the player extra resources through the instance of the ResourceManager, but it also removed the building from play. Collecting the products added those to the player and restarted the production timer.

4.1.6 Team Interface

The greatest amount of work was the team interface. This involved managing the units that the player had produced with their buildings. The team interface, as well as the included unit detail interface, can be seen in Picture 26.



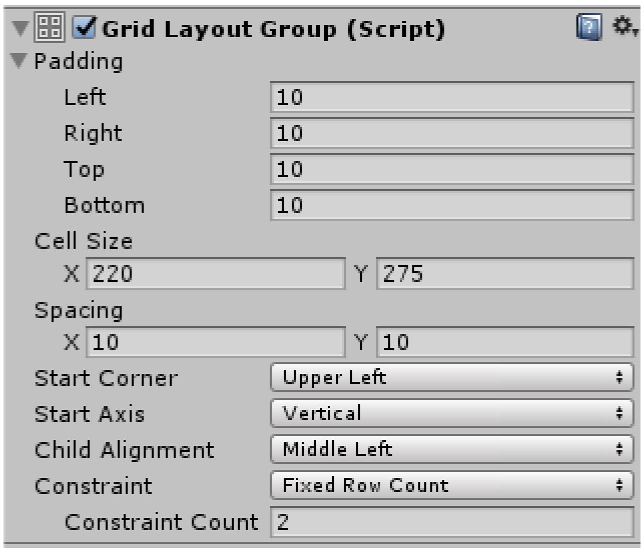
Picture 26. Elements of team and unit detail interfaces for Movenator.

This interface is very much tied to an instance of PenguinManager. PenguinManager contains a collection of the units in the player's possession; these are objects of the Penguin class. These objects can have a main type, which is one of the three categories available, and a sub-type, which can be one of five different sub-types contained in each of the main types. These are used to sort the units in the team interface.

The instance also tracks the player's different teams; there is one for each main type. Each of the teams contains five different units, one of each sub-type of that main type. The PenguinManager script also contains five different events it can trigger: (1) PenguinAdded, which is triggered when the player adds a new unit to their collection; (2) PenguinRemoved, which is triggered when a unit is removed from the player's collection; (3) AddedToTeam, which is triggered when a unit is added to its corresponding team; (4) RemovedFromTeam, which is triggered when a unit is removed from its corresponding team; and (5) TeamLevelUpdated, which is triggered whenever a unit in a team progresses to a new level, increasing the combined team level as well. The TeamPanel class features listeners for all the events.

The interface can be broken down into four parts: the team buttons on the left, the highlighted team on the top, the unit cards in the middle, and the unit detail interface on the right. Each of these had their own implementation, and they can be examined one by one.

Of the four parts of the interface, the unit cards in the middle should be examined first as they are what the whole interface is built around. The unit cards are child objects of a parent which has a Grid Layout Group component attached to it; an example can be seen in Picture 27. These Layout Groups work as the Content of a Scroll Rect component on a Scroll View object. This allows the same kind of behavior that was used for the building cards in the construction interface.



Grid Layout Group (Script)

Padding

Left	10
Right	10
Top	10
Bottom	10
Cell Size	X 220 Y 275
Spacing	X 10 Y 10
Start Corner	Upper Left
Start Axis	Vertical
Child Alignment	Middle Left
Constraint	Fixed Row Count
Constraint Count	2

Picture 27. Grid Layout Group component.

This component is very similar to the Horizontal Layout Group used previously. It is also important to note here that a Layout Group was used again because of the need to support dynamic addition of additional content. Whenever the events PenguinAdded or PenguinRemoved are triggered in the PenguinManager, a new unit card must be created or an old one must be destroyed.

The team buttons on the left were used to select the team to manage and to filter the unit cards accordingly. Pressing one of the buttons would change the highlighted team to the team of the button's category. There were buttons included for each of main types included in the prototype at this time. These buttons also included a number to indicate the current level of the corresponding team; this was updated whenever the TeamLevelUpdated was triggered.

The section of the highlighted team involved several different functionalities. This team, and the unit cards in the middle, was updated to show the desired team any time tapped on any of the team buttons on the left-side of the interface, allowing them to manage that specific team.

Firstly, the symbol for the main type of the team was updated by changing Sprite for the Image component based on the main type choice. Secondly, the buttons including either a plus sign or a unit were updated accordingly. To make the choice, the team of the corresponding main type of units was examined.

If there was no unit of a certain sub-type on the team, an empty button with a plus sign on it was shown for the player. By tapping on a button of this kind, the player could then filter the unit card in the middle to only show the ones that could be used to fill this empty spot on the team.

If there was a unit of a certain sub-type on the team, the image and level of that unit would be shown on the player on the corresponding spot. By tapping on this kind of a button, the player could open the unit detail interface for this unit to remove that unit from the team, making that spot empty again.

Lastly, the highlighted team section needed to display the combined level of the units on that team as well; this was done to make it even clearer for the player. This number also needed to be updated any time the TeamLevelUpdated was triggered, involving the currently highlighted team.

The unit detail interface was implemented to, much like the building production interface, allow the player to further examine a specific unit and use features related to that unit. There were several types of information and functionality that the player needed to be able to access; these included the unit image, name, main type, sub-type, level, competition power, upgrade functionality, retirement functionality, and team management functionality.

Whenever a certain unit card was tapped on, the data required was read from the unit reference that the card had. This data was then used to fill the interface. Likewise, this provided the possibility for the interface to have a reference to the object of the Penguin class so that the unit-specific features could be used by the player.

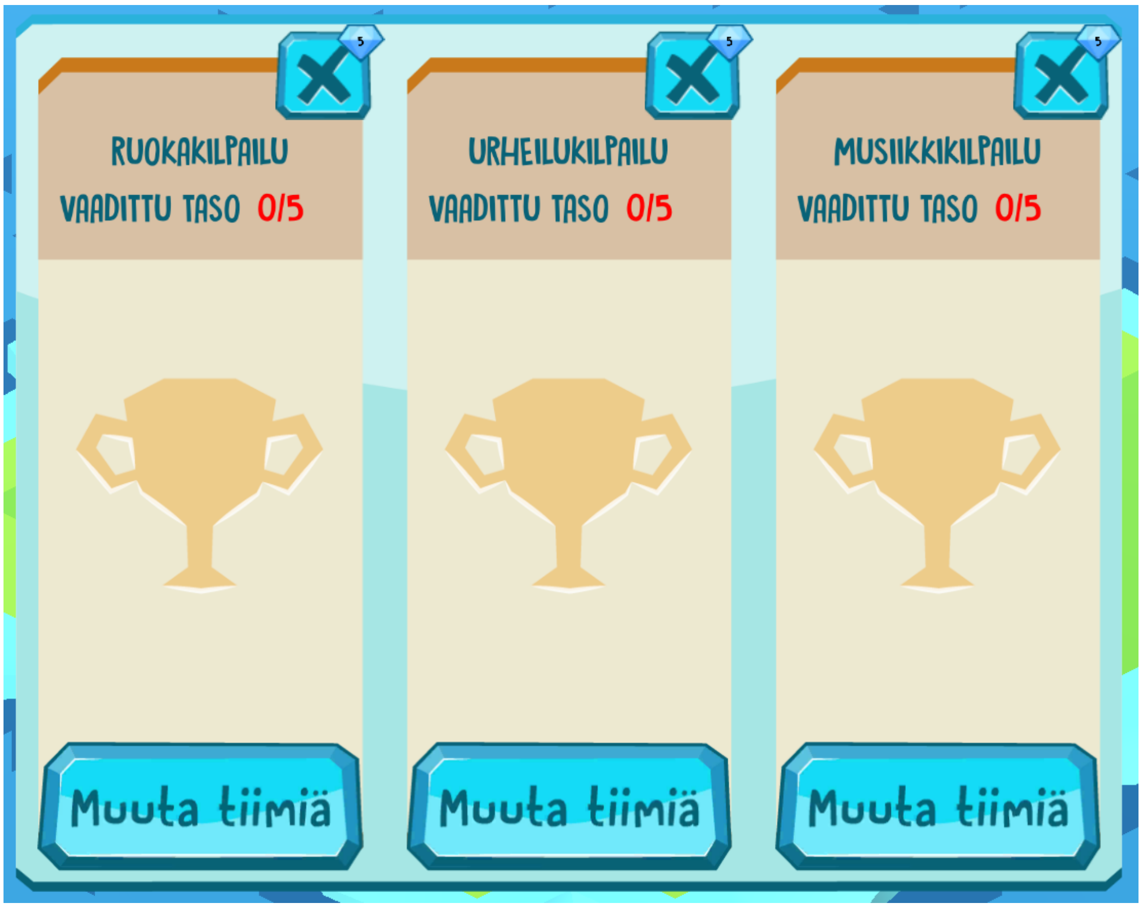
Upon testing the performance of the team interface, a clear problem in performance was noticed. Whereas the framerate would regularly be 60 FPS, it would drop as low as 30 FPS when the interface was opened. This interface also contained a default Scroll View object, just like the construction interface.

Again, the UI Mask component was swapped for a Rect Mask 2D component. The performance was then tested again, and the performance increased to 40 FPS. For the construction interface, the improvement in framerate was 15 FPS. For this interface, it was 10 FPS. It is still a noticeable increase in framerate, and the smaller increase could be because the Scroll View covered a much larger area than in the construction interface.

The framerate of 40 FPS on a low-end device was determined to be good enough for the prototype. However, for future development, the issue should be further investigated to gain an even further increase in performance.

4.1.7 Competition Interface

The competition interface was heavily tied to both the teams the player possessed and the different competition minigames that were built by other developers for the game. The interface offered the player the chance to enter these competitions; the competition interface can be seen in Picture 28.



Picture 28. Elements of competition interface for Movenator.

The competition interface is clearly divided into three elements which perform similar functionality. Each of these competition cards would tell the player the competition's name, required team level to enter, a button to act on the competition, and a button to "re-roll" the competition.

The parent object of the competition interface had a custom CompetitionPanel script component attached to it; this component controlled the actions the player performed on the interface, and generated new competitions for the player whenever they had beaten an existing one.

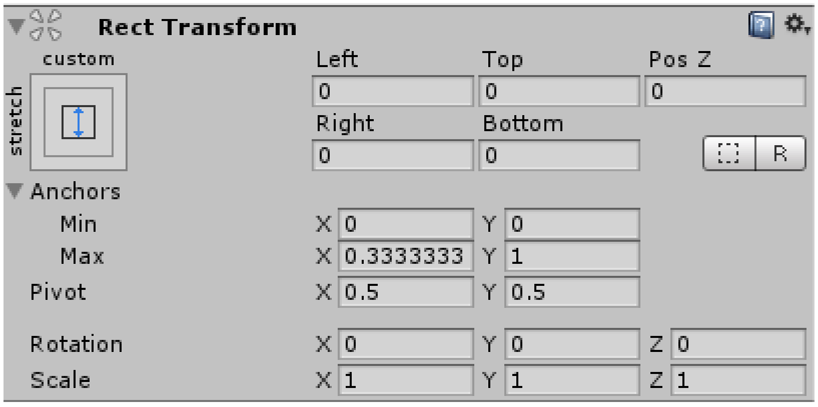
When generated, a competition card would be assigned a type which would be one of the main types of the buildings and units. It would, additionally, be assigned a level requirement. This level requirement was based on the level of the player so the competitions would get more difficult as the player progressed in the game.

Each of the cards also contained a button in the bottom. This button’s appearance and functionality would change based on whether the player’s corresponding team could match the level requirement of the team. If not, as in Picture 29, the buttons would simply direct the player to the team interface to make changes to that specific team. However, if the player’s team’s level matched the requirement, they could use the button to enter the competition.

Additionally, there was a “re-roll” button placed on the top-right corner of the card. This button allowed the player to pay a certain amount of premium resources to discard the currently offered competition and have a new one generated for them. This would be useful in later stages of development as there would be a lot more variance to the types of competitions available.

It should be noted how the three elements were aligned inside the interface. Previously, the use of distinct types of Layout Groups was discussed. However, the reasoning behind their use was always to give future developers a straightforward way to add more content, as the Layout Groups would automatically calculate new position and size values for the UI elements inside the Layout Groups.

For the competition interface, this was not necessary. Instead, a different way of setting up the Anchor values of the Rect Tranform components on the competition card elements was used; an example of one of the Rect Transforms can be seen in Picture 29.



Rect Transform

custom

stretch

	Left	Top	Pos Z
	0	0	0
	Right	Bottom	
	0	0	
Anchors			
Min	X 0	Y 0	
Max	X 0.3333333	Y 1	
Pivot	X 0.5	Y 0.5	
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Picture 29. Rect Transform component determining the position and size of object.

As was also previously mentioned, the use of Layout Group components creates unnecessary processing since every time there is a change (Tantzy Games 2016). Therefore, setting up the Anchor values on the elements’ Rect Transform components can be beneficial to the performance, and it should be done whenever it is possible to do so.

In Picture 29, the focus in on the Anchor values. By setting the minimum Y value to 0 and the maximum Y value to 1, the element will cover the whole vertical height of the parent element. Setting the minimum X value to 0 and the maximum X value to 1/3, the element will cover one third of the horizontal width of the parent element. By determining these Anchor values, the elements can be set to always take an equally large area of their parent element in proportion to the parent element’s other child elements, and they will continue to do so even if the size of the screen changes.

4.2 Mini Golf Universe

For Mini Golf Universe, the UI duties focused much more on a specific interface, the main menu, rather than all the interfaces required for a certain game mode. Due to this, the work overall was done on a much smaller scale. However, there were still many similarities to the development of Movenator. The work for this project was also done later than for Movenator which made it easier to implement more optimal solutions.

4.2.1 User Interface in General

For Mini Golf Universe, many of the same development choices, regarding the general UI options, were made. Here, the Render Mode of the Canvas component of the Canvas object was also set to “Screen Space – Overlay” as the UI implemented was a traditional menu interface that needed to be rendered on top of any objects in the background.

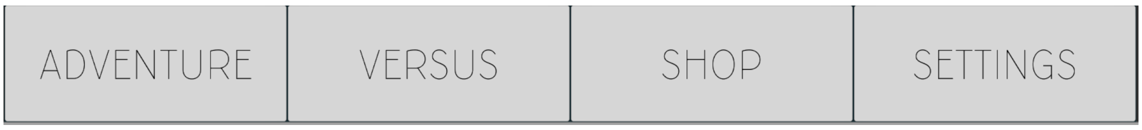
For the Canvas Scaler script component, the same choices were made again; the UI Scale Mode was set to “Scale With Screen Size” and the Reference Resolution was set to 1920 by 1080. This was done because Mini Golf Universe was also to appear in landscape mode, and this resolution was a traditional resolution for mobile devices in landscape mode.

Same steps were also taken to optimized the number of draw calls. A Rect Mask 2D script component was attached to the Canvas object and any unnecessary UI elements were disabled. Therefore, no excessive processing was required. The prototype was tested on a low-end mobile device again, and a framerate of 60 FPS could be reached on most of the interfaces; the performance and optimization of the UI was determined adequate.

The TextMesh Pro components were used in the implementation of different text elements for Mini Golf Universe. As the components were already available at the beginning of the project, it was chosen over the pre-packaged Text component to offer any possible performance gain and to get familiar with the components as they were to replace Unity’s own Text component in the future. (Winter 2017.)

4.2.2 General Menu Interface

The general functionality of the menu is simple. The player has access to four different buttons, shown in Picture 30, that they can use to change the content displayed in the main menu. The process of changing content was implemented somewhat differently for Mini Golf Universe.



Picture 30. Elements of general menu interface for Mini Golf Universe.

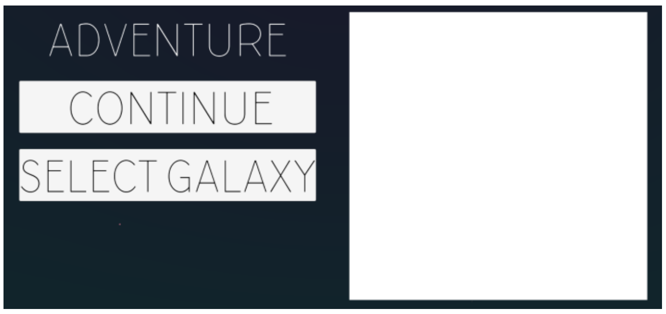
Mini Golf Universe also featured a script called MainMenuManager which was responsible for handling interactions between different interfaces; this worked in a comparable way to the UIManager script in the Movenator project. However, in the Movenator project, the UI Manager object had references of all the different UI elements so that it could enable and disable them. This was partially due to its more complicated UI.

However, in Mini Golf Universe, the Main Menu Manager object did not contain these kind of references and objects were not enable and disable in this script. Instead, a press of a button triggered an event that the objects could subscribe to. Based on the information of the object which caused the event to be triggered, these objects would then enable or disable their corresponding content accordingly. This was found to be a much clearer implementation, especially on the programming side.

4.2.3 Adventure Menu

All three sub-menus of the adventure menu consisted of content that could be set up by using the previously mentioned Anchors of the Rect Transform components. As discussed previously, this was a more efficient method for static content than the use of different Layout Groups which could have calculated the positions and sizes of the elements automatically

For example, the default view, seen in Picture 31, of the adventure menu consisted of a TextMesh Pro element, two Button elements, and an Image element. There was no need to dynamically add or remove content from the view. Therefore, there was no need for Layout Groups. Similarly, both the planet view and the galaxy view would always have the same number of elements on them, making the use of Layout Group components unnecessary.

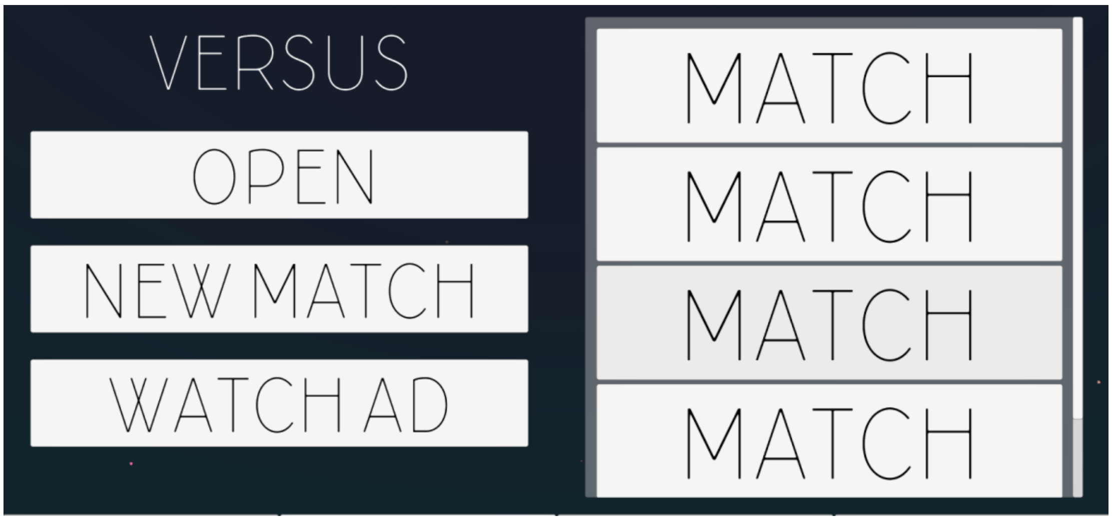


Picture 31. Elements of adventure menu for Mini Golf Universe.

The sub-menus of the adventure menu were controlled by a custom AdventureMenu script. The process of changing between the different sub-menus was done by a method where a parameter of an enum type was used. This kind of parameter was deemed better than the string parameters used in the UIManager script of Movenator project; this is because the enum type offers less room for error than a string value a developer would type in manually, but the enum values can still be descriptive.

4.2.4 Versus Menu

The versus menu, seen in Picture 32, was the interface though which the player would be able to access the online multiplayer features of the game. It featured buttons for opening prizes, starting new multiplayer matches, and watching video ads for prizes; it also contained the functionality for continuing previously started matches.



Picture 32. Elements of versus menu for Mini Golf Universe.

The left half of the menu consisted of simple elements which were positioned on the screen using the Rect Transform components' Anchors. The container for previously started matches on the right was the only part of the whole menu interface that required support for the dynamic addition of content. Therefore, it consisted of a Scroll View object with a Vertical Layout Group component as a part of its content.

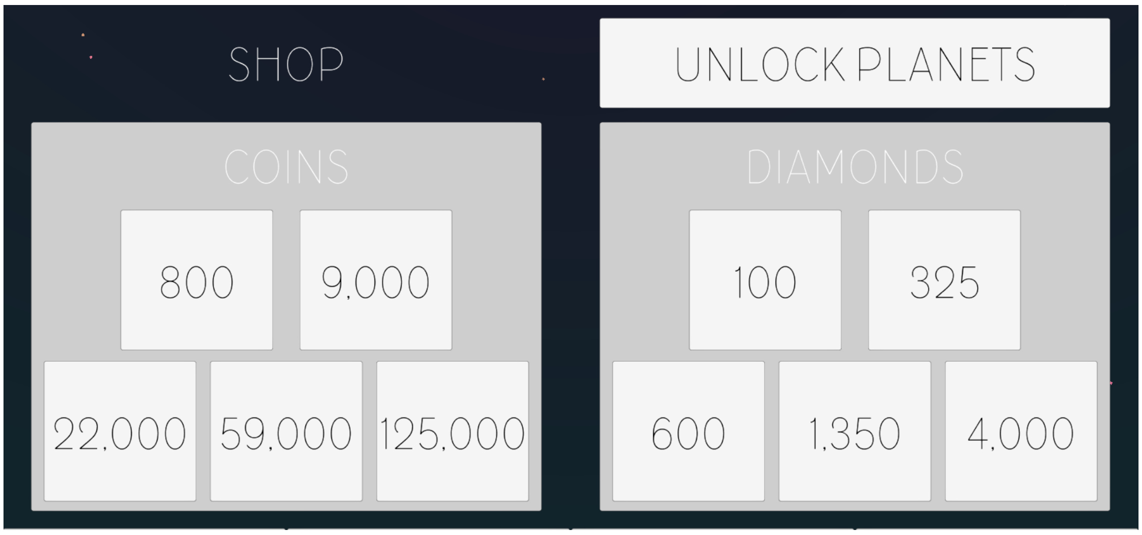
As with the Movenator interfaces that had Scroll View objects on them, the performance was expected to be less than ideal on the low-end test device. Upon testing, the original framerate was only 35 FPS. Several steps were taken to improve the performance.

Firstly, the UI Mask component was replaced by the Rect Mask 2D component as this had improved the performance for the Movenator project as well; this change increased the average framerate to 45 FPS, offering a significant increase in performance. Secondly, the transparent background image, still seen in Picture 32, was disabled; this boosted the performance by further 5 FPS.

By making these changes, the average framerate was increased from 35 to 50 FPS. The final framerate was more than high enough for a menu interface on a low-end device. Thus, no further steps were taken to increase the performance.

4.2.5 Shop Menu

The shop menu, seen in Picture 33, consisted of multiple different buttons. Through them, the player would be able to purchase in-game currencies and additional levels. All the elements were positioned by using the Rect Transforms' Anchor values to offer better performance.

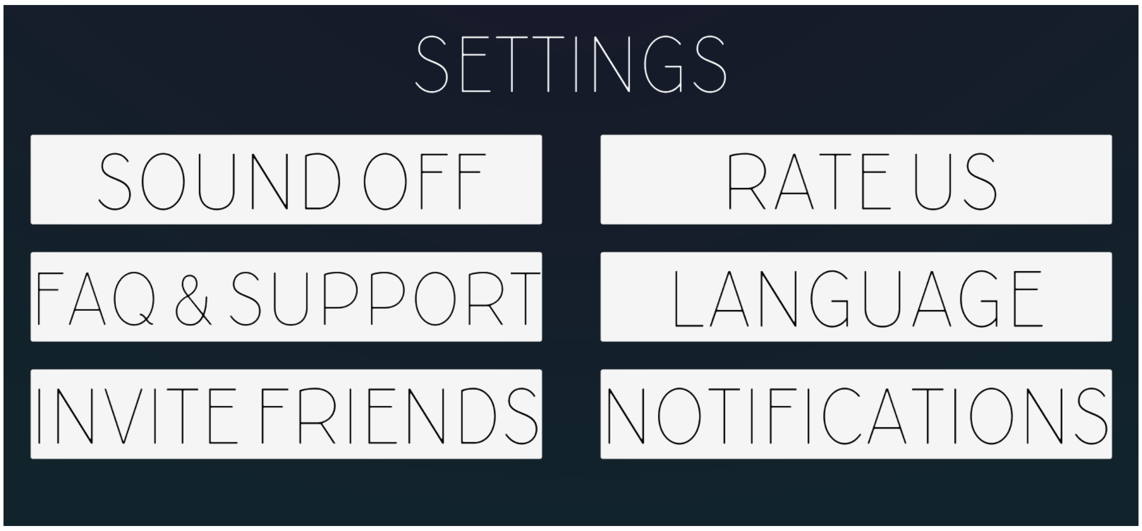


Picture 33. Elements of shop menu for Mini Golf Universe.

The functionality of making purchases also involved communicating with the back-end of the game; this was done to check on the player's available currencies and to also communicate with either Google Play or App Store to make purchases with real money.

4.2.6 Settings Menu

The settings menu, seen in Picture 34, was designed to offer multiple different settings, options, and features that could not be categorized under the other menus. This involved functionality from controlling the audio to changing the notification settings.



Picture 34. Elements of settings menu for Mini Golf Universe.

Much of the menu's functionality had not been decided on yet for the first prototype. Therefore, this menu was built as more of a concept for which the functionality could later be added. The menu elements were again positioned by simply using the Anchor values of the Rect Transform components.

5 FINDINGS

After the implemented functionalities and performed optimizations, the different development choices can be examined; this is done by listing the different items, chosen implementations, and reasons for these choices. This has been done in Table 1.

Table 1. Implementation choices made during development.

Item	**Game**	**Chosen implementation**	**Reason**
		Canvas objects	
Canvas component – Render Mode	both	Screen Space - Overlay	The UI needed to be rendered on top of everything else.
Canvas Scaler component – UI Scale Mode	both	Scale With Screen Size	The UI needed to scale with the screen size.
Canvas Scaler component – Reference Resolution	both	1920x1080	The default resolution for 16:9 aspect ratio. Good for game in landscape mode.
Additional components	Movenator	RectMask2D	Reduced draw calls for active but unseen elements; implemented for future need
		Text objects	
Text components	Movenator	Default Text component	No other choices available at the beginning of development.
	Mini Golf Universe	TextMesh Pro components	An improved component available at the beginning of development; better optimization, default component in the future
		Object positioning	
Static objects	both	achored Rect Transforms	Better performance for static groups of objects
Dynamically added objects	both	Layout Group components	Automatic positioning and resizing of groups of objects when objects are added or removed during the game
		Scroll View objects	
Masking	both	UIMask component replaced by RectMask2D component	Framerate increased by 10-15 FPS (3 implementations)
Further optimization	Mini Golf Universe	Removal of transparent background Image	Framerate increased by 5 FPS (1 implementation)
		Scripting	
Toggling UI elements	Movenator	Method with string parameter	Easy access from anywhere.
	Mini Golf Universe	Events, method with enum parameter	Events for extremely clear and simple implementation, enum parameter offers less room for error and better accessibility.
Dividing script functionality	both	Separate script component for each interface	Increase in code readability

As can be seen in Table 1, there were a lot methods of implementations that were valid for different types of games. For example, it would be safe to say that this specific canvas implementation could be applied to almost any kind of mobile game and its overlay interface. There are, of course, small adjustments that would be needed if the game was played in portrait mode, for example; this would only involve switching the two Reference Resolution values, though.

It is also safe to say that the new TextMesh Pro component should be used for new Unity projects since it will become the default component in the future. It should also offer better performance across multiple platforms and more features as well. The default Text component should only be used in prototyping as it is somewhat simpler to use.

For positioning groups of objects that are static, anchored Rect Transforms should be used; it offers better performance compared to the use of Layout Groups. Through setting up the Anchor values, the same results can be achieved, but the performance cost will be lower. Layout Groups can be used when objects can be added to or removed from the group of objects during the game; this allows the positions and sizes of the objects to be automatically recalculated.

A lot of different Unity tools and components were used in the UI solutions. The Scroll View object was something that caused performance issues every time it was used. However, by simply replacing its UI Mask component with a Rect Mask 2D component often boosted the performance enough. Though, for the team interface of Movenator, the performance could still have been improved further either by improving the Scroll View object further or by developing a custom alternative.

Also, when it comes to implementing a general UI controller script and controlling the other UI elements, it is usually better to use either events or clear parameters such as enums or object references instead of string parameters. Though string parameters are simple to use, they also leave more room for error and often make the code more difficult for future developers to understand. The script functionalities should also be divided into separate scripts by their functionalities. This makes the scripts much easier to understand, and they will also be easier to modify.

A bit over half, 6, of the total interfaces, 10, implemented ran fine after the original implementation. However, for 4 of the interfaces, clear improvements could be made. In Table 2, the different implementations, their performances before and after the optimization process, and the tools used in the process have been listed.

Table 2. Performance improvements.

Interface	Draw Calls or Average Framerate Before Optimization	Draw Calls or Average Framerate After Optimization	Tools used
Movenator			
General	78 draw calls	32 draw calls	Profiler, custom FPS counter
Resource Exchange	60 FPS	-	custom FPS counter
Construction	45 FPS	60 FPS	Profiler, custom FPS counter
Building Production	60 FPS	-	custom FPS counter
Team	30 FPS	40 FPS	Profiler, custom FPS counter
Competition	55 FPS	-	custom FPS counter
Mini Golf Universe			
Adventure	60 FPS	-	custom FPS counter
Versus	35 FPS	50 FPS	Profiler, custom FPS counter
Shop	50 FPS	-	custom FPS counter
Settings	60 FPS	-	custom FPS counter

Generally, the FPS counter was used to check the performance of each interface. Then, the Profiler was used to find any problematic parts of the implementations. After optimization, the performance was checked again using the FPS counter to see if the performance had improved. However, the general game interface of Movenator was an exception; for that interface, the Profiler was used both before and after the optimization to measure the number of draw calls.

Even though the Frame Debugger was planned to be used as a part of the optimization process, no use for it could be found. This could have been because the implementations consisted of graphically simple 2D objects, and the Frame Debugger could be found more useful when dealing with 3D objects which require a lot more graphical processing, for example.

It should be noted again that all the interfaces where clear improvements were made included a Scroll View object. This raises the question how sub-optimal the default implementation of the Scroll View object really is; this is something that could be examined further in the future as well.

6 CONCLUSION

The aim of the thesis was to form a set of guidelines and best practices for the implementation and the optimization of mobile game user interfaces developed in the Unity game engine. This was done by going through the process of implementation and optimization for two different mobile games, Movenator and Mini Golf Universe. In the optimization process, three tools were determined to be used: a custom FPS counter, the Profiler tool, and the Frame Debugger tool.

Most of the initial requirements were filled with the final products, and most of the interfaces' performance was excellent. After the development process, the different development choices were listed and compared between the two projects. From that data, a lot of similarities between the implementations could be found.

The Canvas objects' implementations were extremely similar for both projects, and the implementation of a Canvas object was one of the easiest to generalize. For the Text objects, the implementations varied; however, this was due to the better alternative not being available when the development of Movenator was started.

Clear practices for object positioning methods could be determined. Static groups of objects should use anchored Rect Transforms to achieve better performance. When objects could be dynamically added to a group or removed from it, the different Layout Group components should be used due to their automatic positioning and resizing of the objects in the group.

The default Scroll View object was determined to be very disadvantageous for performance, and it should not be used as is. However, clear steps for optimization could be determined; this involved removing the default UI Mask component of the object. Still, some performance issues remained for some of the interfaces because of the Scroll View objects. In the future, their performance and steps for further optimization should be researched. Developing a custom alternative to the Scroll View object could also be a solution to this.

The custom FPS counter was used a lot during the development. It was used every time the implementations were tested on a real device to determine how well they performed. The counter was also used after adjusting the solutions to determine if the performance had truly improved.

The Profiler tool was actively used as well. Whenever the performance was sub-optimal, the Profiler was used to determine what the cause of it was. Therefore, it was not used as frequently as the FPS counter, but it was determined to be a very valuable tool in the optimization process nevertheless.

The Frame Debugger was not used during the optimization process as no use for it could be found. As discussed previously, the graphically simple interfaces might have been something for which the tool is useless. When moving towards more complex objects, the Frame Debugger might prove to be more useful.

REFERENCES

Brightman, J. 2017. Mobile games booming as global games market hits $108.9B in 2017 - Newzoo. Gamesindustry.biz. Consulted 2.6.2017. http://www.gamesindustry.biz/articles/2017-04-20-mobile-games-booming-as-global-games-market-hits-usd108-9b-in-2017-newzoo.

Dring, C. 2017. App Store revenue breaks $70bn. Gamesindustry.biz. Consulted 2.6.2017. http://www.gamesindustry.biz/articles/2017-06-01-app-store-revenue-breaks-usd70bn.

Fear, E. 2009. United They Stand: How Three Guys Found Unity in Starting a Revolution. Develop. Consulted 6.4.2017. http://www.develop-online.net/analysis/united-they-stand/0116643.

Flick, J. 2015. Frames Per Second. Catlike Coding. Consulted 9.5.2017. http://catlikecoding.com/unity/tutorials/frames-per-second/.

International Data Company 2017. Smartphone OS Market Share, 2016 Q3. Consulted 6.4.2017. http://www.idc.com/promo/smartphone-market-share/os.

Tantzy Games 2016. Optimizing Unity UI. Consulted 28.5.2017. http://www.tantzygames.com/blog/optimizing-unity-ui/.

Thorn, A. 2013. Learn Unity for 2D Game Development. New York, NY, USA: Apress Media.

Unity Technologies 2017a. Company Facts. Consulted 6.4.2017. https://unity3d.com/public-relations.

Unity Technologies 2017b. Unity Store. Consulted 6.4.2017. https://store.unity.com.

Unity Technologies 2017c. Multiplatform. Consulted 6.4.2017. https://unity3d.com/unity/multiplatform.

Unity Technologies 2017d. Unity Manual. Consulted 6.4.2017. https://docs.unity3d.com/Manual/index.html.

Unity Technologies 2017e. Scripting API. Consulted 6.4.2017. https://docs.unity3d.com/ScriptReference/index.html.

Unity Technologies 2017f. Profiler overview. Consulted 9.5.2017. https://docs.unity3d.com/Manual/Profiler.html.

Unity Technologies 2017g. Frame Debugger. Consulted 9.5.2017. https://docs.unity3d.com/Manual/FrameDebugger.html.

Unity Technologies 2017h. Canvas. Consulted 28.5.2017. https://docs.unity3d.com/Manual/UICanvas.html

Unity Technologies 2017i. Canvas Scaler. Consulted 28.5.2017. https://docs.unity3d.com/Manual/script-CanvasScaler.html

Unity Technologies 2017j. RectMask2D. Consulted 28.5.2017. https://docs.unity3d.com/Manual/script-RectMask2D.html

Unity Technologies 2017k. Button. Consulted 28.5.2017. https://docs.unity3d.com/Manual/script-Button.html

University of Turku 2017. Movenator. Consulted 2.6.2017. https://www.utu.fi/en/units/med/units/hoitotiede/research/projects/digital-nursing-turku/tepe/Pages/Movenator.aspx.

Unreal Engine 2017.Frequently Asked Questions (FAQ). Consulted 25.5.2017. https://www.unrealengine.com/faq.

Winter, B. 2017. TextMesh Pro Joins Unity. Unity Blogs. Consulted 1.6.2017. https://blogs.unity3d.com/2017/03/20/textmesh-pro-joins-unity/.