Hans-Jürgen Krupp

Harvest Survive

Game Mechanics of Unity 2D Game

Helsinki Metropolia University of Applied Sciences

Degree Bachelor of Engineering

Degree Programme Information Technology

Thesis

Date 25.9.2014

Helsinki Metropolia University of Applied Sciences

Author(s) Title Number of Pages Date	Hans-Jürgen Krupp Harvest Survive : Game Mechanics of Unity 2D Game 70 pages + 0 appendices 25 September 2014
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor	Juha Huhtakallio

The purpose of this project was to learn how to create Games in Unity 2D, to see the workflow and to test if the new Unity 2D feature of the Unity engine was a good alternative for developing 2D games. A further aspect was to learn the different steps and mechanics of the Unity environment.

The goal was to create a survival game, in which the player would have to grow plants in order to get food and money to stay alive in a hostile environment. The player has to survive in six different areas with different monsters and one final boss in the last area. The player wins the game when the last boss is killed. To achieve this, the player has to grow a large number of crops and survive the roaming monsters and earn money to get the best equipment.

As for the technology used in the project, the game engine Unity was used with the plugins Toolkit2D for the tilemaps, Navmesh2D for the pathfinding of the monsters and the Necromancers skin for the user interface.

The result of the project was a working Unity 2D game, which can be played on Windows systems. Moreover it is the first step into making Unity games and provides useful experience for future Unity projects.

Keywords	Unity, 2D, Game, Navmesh2D, Toolkit2D

Contents

1 Introduction

The project is a survival game with a focus on planting and harvesting crops while defeating monsters. This was made with Unity 2D and the target platform was Windows. The project was a single-person project. Moreover the topic was chosen because Unity is becoming more famous among Indie developers because of its free license, which can be used to publish games until an annual gross revenue of 100,000 $. [1]

Of course not all features are included in the free version, but it is enough for Indie game development, especially if it is used for 2D development. Moreover Unity focuses mainly on portability, which means it is possible with the same project to release it to Windows, Linux, Mac, Web, Android, Ios, Windows phone, Blackberry, Xbox360, Xbox One, Playstation3 and Playstation4, without a big effort for porting it manually. This makes it easy to reach more customers and in the end it means more money for the developers. [1]

Unity 3D is a well known game engine and has been released for many years, but it has never supported 2D development. It changed when they released Unity 2D at the end of 2013, so it was interesting to get started with it and see its advantages and disadvantages in making a 2D game with it.

In the plot of this game the goal is to plant crops and wait until the plants grow, then harvest them and use the grown fruits and vegetables as food or sell them at a store to earn money. With the money the player can buy new seeds, equipment and weapons. The equipment and weapons are necessary to fight the monsters which roam the areas. In the last area there will be a boss monster which has to be defeated in order to win the game. The game will be lost when the player dies because of receiving damage from monsters or losing health from starving.

2 Theoretical Background

2.1 Overview

In this section I will briefly explain the history of Unity and the development and reasons for the success of the game engine. After that I will briefly discuss the features of the Unity game engine in the focus of the 3D technology. At last I will present the new features of the Unity 2D engine, which was released at the end of 2013.

Most of the information regarding the Unity engine I found on the official Unity webpage, as in literature the authors usually just reference to the webpage for more and current information. Additional source material I found on the developers' webpages of the tools I used.

2.2 Unity

2.2.1 History

In this chapter I will briefly tell about the history of Unity after a short explanation of what Unity is. Unity is a cross-platform game engine and integrated development environment (IDE) developed by Unity Technologies. The company was founded in 2004 by David Helgason, Nicholas Francis and Joachim Ante in Copenhagen, Denmark after their first game failed to be a financial success. They noticed the value in creating tools and an engine to create games. Additionally they wanted to make their engine affordable for Indie developers and any other game programming enthusiast. Their project received funding from big investment partners and gained success because of its support for independent developers who were unable to either create their own game engine or purchase one. [2]

The first release version of Unity was launched at Apple's Worldwide Developers Conference in the year 2005. At that time it was built only to function and build projects on the Mac platform but still got enough success to continue the development of the engine and tools. The next release with Unity 3 was in September 2010 and focused on introducing more tools, which high-end game studios usually need for their projects. This made bigger development studios interested in Unity, but smaller ones as well because they provided a game engine with an affordable price. [2]

Unity continued developing since that day and with the introduction of the Unity version 4.3, support for 2D games started at the end of 2013. This version came with a large amount of new features especially concentrating on the new 2D area. The latest version 4.5, which is the latest version at the moment, was released last summer 2014. However already this autumn Unity Technologies wants to release the next generation with version 5.0 with a large number of new features to come. [1]

2.2.2 Features

There are a large number of features in the Unity game engine so I will only briefly explain them. The Unity editor is one of the core components of the engine. This is the visual user interface which connects the coding with the assets and so makes it possible to easier see the result of the coding directly in the interface.

The most common views in the Unity editor are the project browser, which shows all assets in a list, which makes it easy to keep track of all the assets in the current Unity project. Another important view is the Inspector, in which it is possible to change properties of game objects in the scene and assets in the project. Moreover the scene view is the sandbox to create the game in. The game view is as the name says the view where it is possible to start the game directly in Unity and preview how it will look and run on the target devices. The last introduced view is the hierarchy window, which shows all game objects of the scene. [3]

Another feature for the Unity editor is that it is possible to extend it with one's own editor tools and build them directly into the Unity editor interface. That means that it can be extended with new functionality added to the existing views or with new customized windows and inspectors. The new editor windows can be moved and docked like any of the Unity's windows. Custom inspectors make it possible to completely control how users view and edit the custom components. This can be used for debugging purposes or increasing productivity in the workflow. [3]

Moreover in Unity it is easy to import models, textures, audio, scripts, sprites and other assets into the Unity project. The assets will be automatically imported when they are saved in the project folder. In addition assets can be modified at any time and the change can be seen immediately in the game. Additionally Unity can import 3D models,

bones and animations from almost any 3D program, for example from Maya, 3ds Max, Modo, Cinema 4D, Cheetah3D or Blender. [4]

Unity also features the height-map to normal-map conversion, which means that any texture can automatically be converted into a normal-map, even when the image files are changed later. Moreover Unity supports several different mipmap generation methods like detail fade, Kaiser filters and gamma correction. Moreover multi-layer photoshop files are automatically compressed in Unity and Unity includes five different presets to quickly set up the textures. Furthermore it is possible to override size and compression settings for each platform, so one source file is enough for all the different platforms. [4]

Moreover Unity can import any audio format which is supported by FMOD. This ensures audio is consistent between all of Unity's supported platforms. For reducing the file size it can be internally converted into Ogg Vorbis. Another feature is the native support for allegorithmic substances. These are hybrid assets, which can have multiple outputs to generate complete texture sets based on the same set of parameters. With use of the substance importer class, it improves the substance-related workflow with asset post-processing and it gives direct access to the imported procedural material instances. [4]

Another feature is the new project browser. With it it is possible to search and preview assets in large projects. Additionally favorite asset searches can be saved and it is possible to bookmark all important folders and pre-defining categories of assets. Furthermore it is possible to preview free and paid content from the asset store in the project window. The asset store is integrated in Unity where artwork, editor plugins, scripts and many other things can be directly downloaded and installed into Unity. Additionally Unity offers a large number of tutorials to get started with. [4]

In every Unity scene empty game objects, which are empty containers, are added. It is possible to add components to the game objects to add functionality such as light, physics, audio, cameras and particle effects. Their core values can be changed directly in the inspector and for more control they can be changed with scripts. [5]

Another function is that components can easily be added to game objects with a drop-down button, and with a copy and paste ability it is easy to move components between

the game objects. For using complex objects repeatedly, they can be turned into a prefab, which can be put anywhere in the game or accessed with scripts. The advantage is that if this original prefab is changed, all other prefab objects in the scene will be changed as well. Other objects such as spheres, boxes, capsules and meshes can be positioned, scaled and rotated and created directly in the Unity editor. There are many features such as grid, surface and vertex snapping tools, which ensure that the positioning of the game objects are correct. [5]

Moreover Unity has with the “Play” mode a development tool for fast editing while playing the game. In the Unity editor, if the play button is pressed, the game starts immediately and gives a preview of the real game which is developed. It is also possible to alter values while testing the prototype and see the results immediately. As well Unity has tools for debugging every frame of the game and with the profiler showing the parts of the game which have the biggest impact on the game performance. It reports which areas of the game most time is spent on and it can be used to find and remove bottlenecks which cost a large amount of performance and make the game run slowly. This tool can be used on any development platform. [6]

For scripting Unity offers three languages to use, C#, JavaScript or Boo. Any of those languages is supported and they run on the Open Source .NET platform Mono. However it is not possible to use scripts to call each other, so it is best to use just one of them throughout the game project. MonoDevelop has the standard debugging features like most IDEs. To change game objects in the game it is important to reference them from the game scene in the Unity editor with the scripts. Then values of objects such as scale or position can be easily accessed and changed. The referencing can be done by name, tag and type of the object. [7]

Another important part is networking, which is integrated in Unity and features real-time networking which can be accessed with only a few lines of code. For synchronization between game objects values, Unity uses a delta compression algorithm or uncompressed unreliable strategies. Moreover the .NET socket libraries are used for real time networking and to open TCP/IP sockets or send UDP messages. It is also possible to access databases with those libraries. Furthermore if the game is run in a browser, the Unity web player communicates with the container web page and has JavaScript and AJAX capabilities. Regarding networks Unity supports also remote procedure calls, which is an essential networking feature. [8]

Unity supports Windows DirectX 11 graphics API, which improves performance with computing shaders, makes it possible to use GPU as a parallel CPU and supports shader model 5.0, which allows the use of more complex shaders. Moreover Unity uses light pre-pass technique for its lighting, as well as linear space lighting and HDR rendering. Additionally Unity comes with a variety of 100 shaders from simple to very advanced ones. Unity offers also access to the GL class in Unity, which is a low-level graphics library with which active transformation matrices can be changed and rendering commands can be issued similarly to OpenGL. Another graphics feature is surface shaders which help with rendering the graphics on multiple devices and ensures it scales correctly. Additionally Unity offers occlusion culling tailor, which reduces the number of rendered objects. [9]

The Unity lightmapping tool bakes lights into textures, which increase the performance and it is possible to just bake parts of the scene, which is currently in use. Moreover dual lightmapping is used, which uses one lightmap for distant scenery and a second for only bounce light to improve the performance. Furthermore there are many additional light effects, which can be baked into the scene as well as light probes, which bake lighting onto moving objects. Moreover Unity supports high-quality real-time shadows for all types of lights. [10]

Unity uses special effects such as depth of field and motion blur, which are optimized for DirectX 11 and post-processing effects such as Edge Detection, Bloom, Vignetting, Tonemapping and Color correction. Another special effect is Render to Texture effect, which makes it possible to add images and dynamic camera content to any surface. Additionally it is possible to use special effects to create reflective or refractive water surfaces. [11]

As particle system Unity uses Shuriken, which is a curve and gradient-driven modular particle system tool. Moreover it gives the functionality for world collision such as Bent normals, automatic culling and external forces, which can be used for example for hurricanes, fireworks, explosions etc. [11]

Another tool in Unity is able to carve, raise and lower terrains. Additionally with this tool it is possible to set up the trees within the Unity editor and to add branches, twigs and leaves, which can be previewed in real-time in the editor. Moreover Unity will put all

textures together into an atlas and automatically calculate ambient occlusion and wind factors for the trees. [12]

For audio Unity uses FMOD, one of the world's most widely used libraries and toolkits. With this it is possible to preview the sound directly in the Unity editor and adjust it to the needs of the developers. Additionally it offers DSP filters, which make the sound more realistic. It has also some advanced features such as lowpass and highpass filters, distortion filters, chorus filter, echo filter and reverb filter. Moreover to make the distribution size smaller, Unity uses the most common tracker file formats such as MOD, IT, S3M and XM. One of these module files can contain many samples or patterns, without taking a large amount of space and all filters can be used on them. If ogg video and audio files are used, they can be streamed from the net, which reduces the size of the web player. [13]

Another feature Unity offers is the 3D physics engine NVIDIA PhysX, which makes it possible to simulate correctly moving hair and clothes, explosions and other physical effects. This system makes use of rigidbodies, joints and colliders which simulate the physics effects. It features also a ragdoll wizard with which it is possible to implement a full ragdoll from an animated character. [14]

Moreover Unity has an integrated pathfinding system, which automatically generates a NavMesh. Those describe the borders of any navigable space in the game and at runtime calculate the paths for the game objects. Since Unity 4 it is possible to use also NavMesh obstacles, which react to changing environments in runtime. [15]

For animations Unity uses Mecanim, which is a flexible animation system that makes fluid and natural motion possible. This animation system is integrated in the Unity engine, so there is no need for 3rd party tools. With this tool it is possible to produce muscle clips, blend trees, state machines and controllers directly inside Unity. Additionally Mecanim can be used to animate many different elements such as sprites, blend shapes or light intensity. Additionally there are two different ways to use animation clips. The first is a multilevel blend tree, which makes it possible to create a wide variety of motions from just a few motion clips. The second is the hierarchical state machine, which defines with conditions when each state is being executed. It is also possible to have several different state machines as layers which makes it possible to have more complex conditions for certain animations. Moreover in Unity there are IK

rigs which are automatically generated to adjust for example feet on the ground and hands on a ledge. [16]

2.2.3 Unity 2D

After listing all the main features of Unity 3D I will list the new features of the 2D part of Unity which were released at the end of 2013. With this release Unity supports the creation of 2D games natively for the first time without the need of using 3rd party tools. One of the new features is the automatic sprite splicing, which makes it easy to slice a spritesheet into its single sprites. This can be done also manually, which has to be done in certain cases. Another new feature is the 2D physics system, which uses the same system of rigidbodies, joints and colliders as in the 3D solution but is driven by Box2D, which is the physics engine making all the 2D physics possible. It is also possible to mix 2D and 3D physics without getting problems or using 3D models in the 2D environment. [17]

Moreover the animation system Mecanim now supports also 2D animations and with its state machine makes it an important tool for any moving 2D characters and objects. Furthermore it is now possible to easily create animation clips by using several sprites and then adjust them with the Animation editor. Additionally in the Unity editor is an extra 2D view, which makes it easy to edit and create 2D Objects. [17]

2.3 Summary

After introducing the Unity game engine, its features and the new 2D capabilities in the next chapter I will show the additional tools I used for my project and then the project itself. This project has been chosen to examine the capabilities of Unity in making a 2D game with the new introduced 2D tools it offer and to get experience with an often used game engine, which can help work in future projects in the game industry.

3 Technology Presentation

The technologies used include the Unity game engine with MonoDevelop as integrated development environment (IDE), 2Dtoolkit for helping with the tilemaps, in other words

creating the game map and Navmesh2D for the pathfinding of the monsters. Additionally Graphics have been mainly used from RPG maker VX Ace and its downloadable contents. Moreover for the Graphical User Interface (GUI) skin the Necromancer GUI from the Unity asset store has been used.

3.1 2Dtoolkit

2Dtoolkit has been developed by Unikron Software, which is an independent software developer based in Newcastle in England. They have released, until now, only this plugin for Unity and before it has been an important tool for being able to create 2D games with Unity 3D. Now with the new Unity 2D features it is still a good tool, which helps with the development of 2D games and offers several features, which Unity 2D is not capable of. The most important feature is to be able to create tilemaps. This makes it easy to create a map without having to be afraid that the sprites were overlapping each other. [18]

This plugin also gives the complete C# source code, so it makes it possible to tweak the tool to the needs of the developers. Another feature is creating sprites, which can have platform-specific size. This is practical for publishing on several platforms. The plugin alos has an own camera tool, which can help with solving resolution and aspect ratio problems. Additionally the plugin offers to slice, tile or clip sprites and also offers dicing of large images, which cuts them into small pieces, helping the performance. Moreover another feature is box and custom shape colliders which will work in the future with the Unity 2D physics. Furthermore 2Dtoolkit offers atlasing, which rebuilds sprite collections for better performance. [18]

Another feature is a static sprite batcher, which can merge large numbers of sprites and colliders into one mesh. This is useful for static backgrounds. The 2Dtoolkit also has sprite animations, which can be used with any sprite collections. Moreover this plugin offers the use of User Interface (UI) components. However the most important feature for my project was the tilemap editor. With this tool it is possible to directly paint in the Unity editor sprites, such as in a paint program. This makes it very easy to create a map. Moreover the tiles in the tilemap can be exchanged with prefabs. [18]

3.2 Navmesh2D

Navmesh2D is a pathfinding plugin for Unity developed by Pigeon Coop. It is a new game company which does not have its own homepage yet. They developed the plugin for their own game and then released it to the Unity asset store. This plugin is a tool to generate and navigate navmeshes for 2D projects, which works similarly to the built-in navigation system Unity 3D uses but which cannot be used for 2D projects. [19]

4 Harvest Survive

4.1 Basic Structure

The game project took half a year to finish and it was a large amount of work to get used to Unity. For the game I used mainly JavaScript, which I would call Unityscript, because JavaScript has many different versions unlike C# which has a fixed library and syntax. Even my last game project was written in JavaScript. It took me some time to learn the new syntax used in Unity. Moreover I used mainly Unityscript for the programming and a little bit of C# for the path finding scripts. This was necessary because the Navmesh2D tool is written in C# and it did not have any JavaScript converter.

First I will give a short overview of the structure of the game.

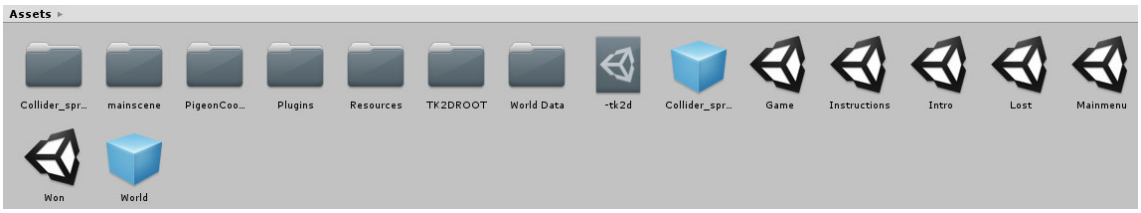

Figure 1. The structure of the “Assets” folder.

Figure 1 shows the structure of the game. The first folder “Collider_sprites_collectiondata” has the sprite collection data for the collider tilemap, which is used for the Navmesh2D and path finding. This is connected with the collider sprite collection prefab. The next folder with the name “mainscene” saves the Navmesh data for path finding in that folder. “PigeonCoop” folder has the code of the Navmesh2D tool. The “plugins” folder and the “TK2Droot” are both folders for the 2DToolkit and contain the source code. The “Ressources” folder contains all graphic files, animations and my own game scripts which are used in the game. The “World Data” folder contains the

spritesheet data for the main tilemap and is connected with the world prefab. The “tk2d” file has data about the 2Dtoolkit tilemaps used. The items with the Unity sign are Unity scenes. One of them is the main scene “Game” which contains everything related to the game. Additionally “Instructions” is the scene, which explains how the game is played. Then there is “Intro”, which is the first introduction screen after starting the game from the main menu. Additionally “Lost” and “Won” are the scenes for loosing and winning and “Mainmenu” is the main menu scene.

Next I will show the structure of the “Resources” folder.

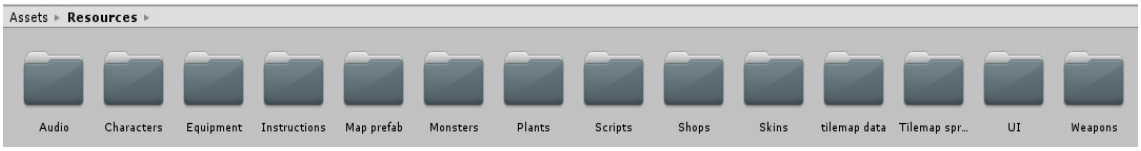

Figure 2. The structure of the “Resources” folder.

Figure 2 shows the structure of the “Resources” folder. The “Audio” folder contains all the sounds and music which is used in the game. The next folder “Characters” has the character sprites inside and its animations. Moreover the “Equipment” folder contains all the animations and sprites for the equipment. The “Instructions” folder includes all the pictures for the instructions in the game. Then the “Map prefab” folder contains map prefabs such as collider prefabs for the tilemaps. Additionally the “Monsters” folder contains all the monster sprites, prefabs and animations. The “Plants” folder includes all the plant sprites, icons, vegetable sprites and icons. The “Scripts” folder contains all the self-written code for the game. The next folder “Shops” has all shop sprites and the shop prefab. Next is the “Skins” folder, which contains all GUI skins for all GUI interfaces such as inventory, character, equipment, status, in game options menu, intro, won screen and lost screen. The “tilemap data” folder contains data for the tilemaps, all the sprites which have been painted and the positions. The “Tilemap sprite collection” folder consists of all the environment spritesheets which are used for the game. The “UI” folder contains some UI sprites such as the health bar for enemies and finally the last folder “Weapons” contains all weapon sprites, animations and prefabs.

4.2 Map

In the game there is only one big map which is divided into six different zones. These zones have all different terrains, monsters and shops. For creating this map I used the

2Dtoolkit to help with the creation of tilemaps, which is not possible with Unity 2D. For using the tilemaps first I had to create a tilemap sprite collection and add the spritesheets I wanted to use for the map.

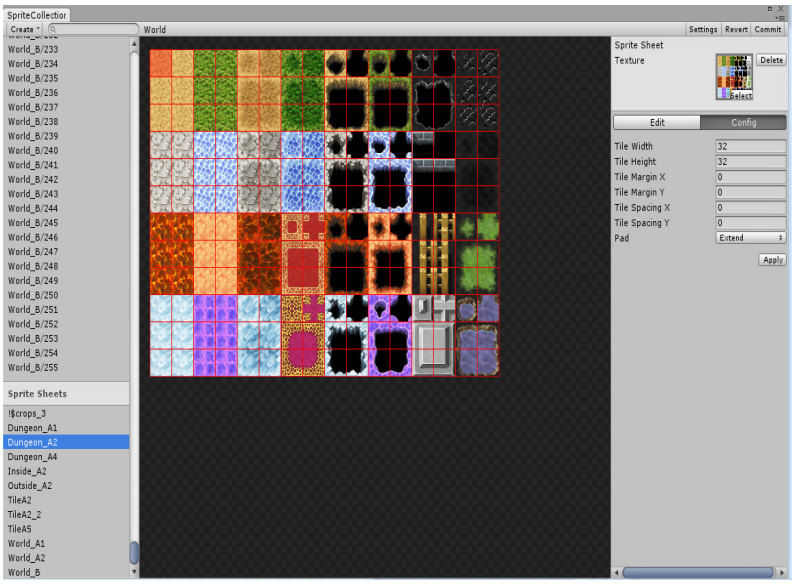

Figure 3. The “SpriteCollection” window.

Figure 3 shows the “SpriteCollection” window with the settings of the imported sprite sheet. On the right side it is possible to choose the width and height of the tiles, so that the tool can cut the spritesheets into single sprites with a fixed value. Moreover in my game all tiles are of the size of 32 width and 32 height. This has to be the same value in the whole tilemap. Otherwise it will not fit and there will be some problems. After choosing the value it was important to click the commit button. Otherwise the new sprites will not be saved in the sprite collection. After adding all the necessary sprites it is possible to move to the editing part of the tilemaps.

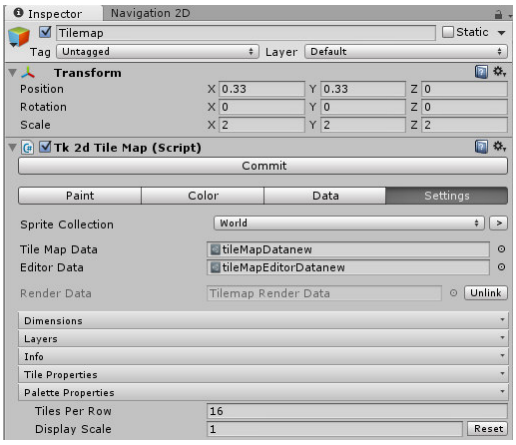

Figure 4. The “Tilemap” in the Unity editor.

Figure 4 shows the tilemap settings and that it is important to add the tile map data, the tile map editor data and the sprite collection, which was added before. The tile map data contains the data, position of the single tiles and prefabs and the map editor data saves the editor preferences. [20]

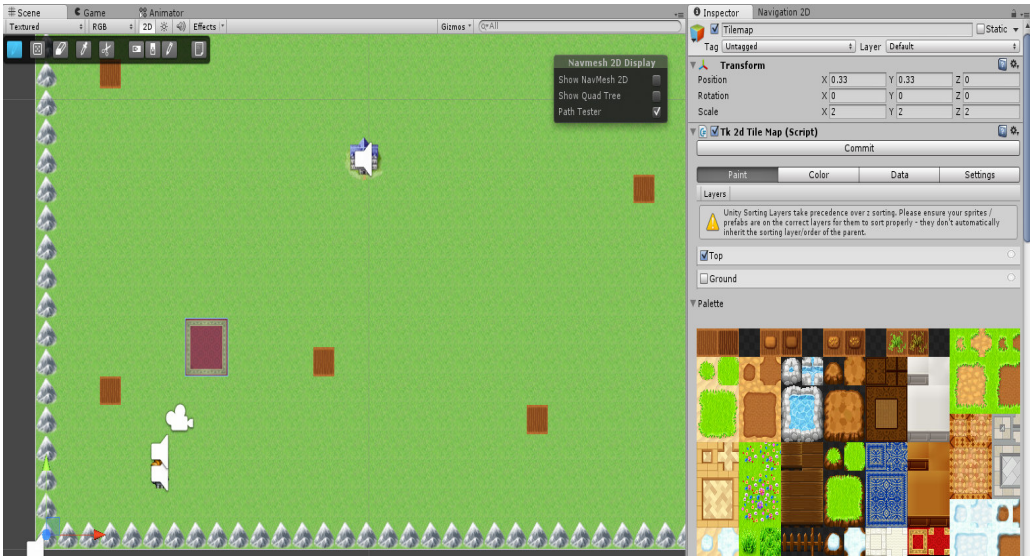

Figure 5. The Paint window of the tilemap.

Figure 5 shows the painting process of the tilemap. On the right side there are the sprites added to the sprite collection and can be easily marked with left click and then painted on the scene on the left side. It is also possible to delete sprites just with marking the delete function on the top left. This makes it quite easy to paint sprites to the tilemap. Additionally it is also possible to replace a sprite with a prefab, which is very useful for making blocked tiles such as the mountains on the scene in figure 5.

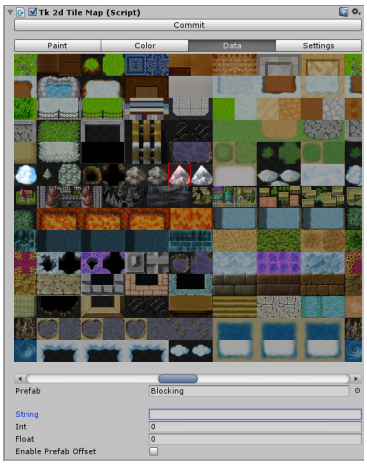

Figure 6. Declaring prefab in tile map.

Figure 6 shows how to declare a prefab for the saved sprite so all sprites of this type will be replaced by the defined prefab. In this case I replace the mountain sprite with the prefab Blocking. This prefab has a boxcollider2D, which is a physics component in Unity and makes it possible to block other objects from moving through the box area, which can be defined at the blocking prefab. With this the map has tiles which cannot be walked on by the player and monsters.

On the map I use several prefabs, one for blocking player movement and monster movement as explained above then plant fields, which are fields the player can plant seeds and harvest crops on. The last one is the gate prefab, which is basically a blocking prefab, so it blocks the path of monsters and the player. However if a boss monster is killed, the gate will open and the player can move on to another area.

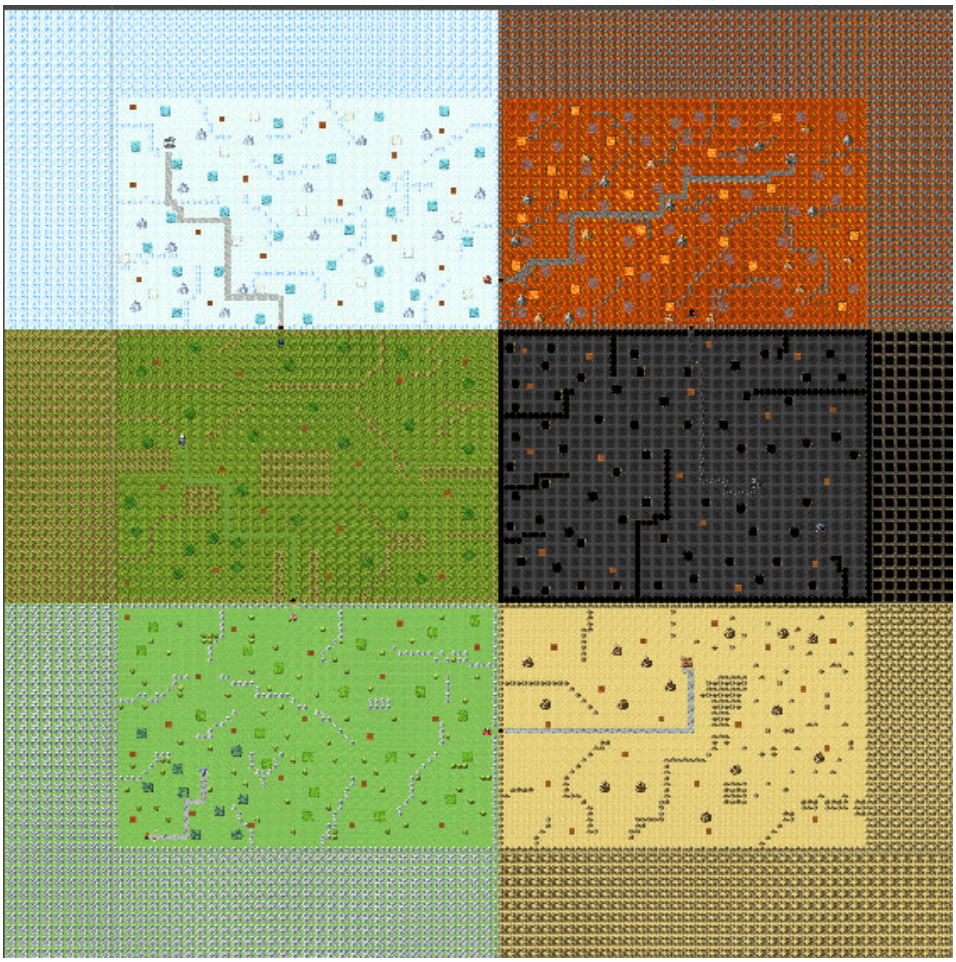

Figure 7. The Map of the game.

Figure 7 shows the ready map of the game with the easily visible six different zones. The starting area is marked as level 1 area and is the grass area at the bottom left. Then there is the level 2 area, a desert, which is on the bottom right. Additionally there is the level 3 area on the left which contains forest. The next area, level 4, is the snow area, which is on the top left. Moreover the second last area is the fire area, which is level 5 and then there is the last area, the shadow area, which is level 6 and contains the final boss of the game.

4.3 Player

The player is one the most important parts of the game, so I will explain in detail the most important features such as inventory system, character system, equipment system, movement, weapon system and attack system. At first I will explain the simple movement for the player.

4.3.1 Movement and Animations

The movement is done in the “Mainchar” script, which has most functionality related to the player. Additionally it is important to have an Animator as a component for the player, it takes care of the change of the animations, which are simply three sprites which change fast. Another component which is needed is an animation controller which has to be placed into the animator that Unity knows the conditions of changing animations. The animation controller has to be edited with the “Mecanim” of Unity which is a state machine and it is possible to define each possible state with conditions.

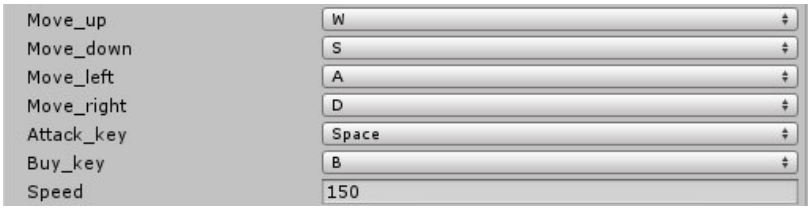

Figure 8. The public movement variables.

Figure 8 shows the public variables for moving the player and also the key for attacking and buying and the speed of the player.

Figure 9. The Animator of the player.

Figure 9 shows the animator for the player and the controller which is also called player.

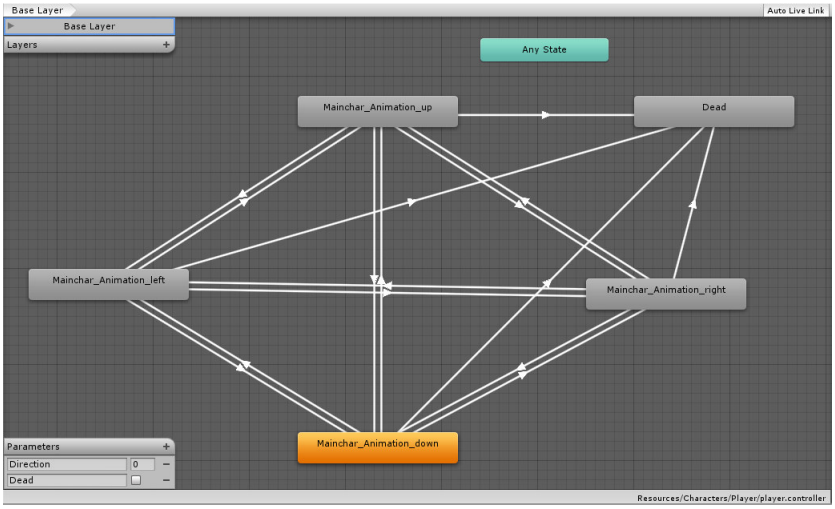

Figure 10. The Mecanim animation tool from Unity: Player movement.

Figure 10 shows the different states of the player. There are five different animations, one for each direction and one when the player dies. Of course each of those states has to be assigned an animation clip which contains the animation. The lines with arrows show in which direction the state can change and on the lines itself there are the conditions for changing the state. Additionally for the four directions there is the Direction variable which is an integer and has the values from zero to three. Each number is for one of the movement states. Moreover there is a Boolean “Dead” which is true, when the character is dead and then the dead animation will be played.

Now I will show the code in the “Mainchar” script relating to the movement and change of the animations.

```
140     if(Input.GetKey(move_up)){
141
142     last_key = "up";
143
144     rigidbody2D.velocity.y = speed * Time.deltaTime;
145
146     rigidbody2D.velocity.x = 0;
147
148     animator.SetInteger("Direction", 2);
149
150         if(has_weapon == true){
151         //change weapon in right direction
152         Current_weapon.GetComponent(Weapons).turn_weapon("up",transform.position.x+0.4,transform.position.y+0.1);
153         }
154     }
```

Figure 11. Moving up and weapon position of the player in the “Mainchar” script.

Figure 11 shows the movement code for the player. In the first line it is checked if the `move_up` button is pressed. If it is true, `last_key` will be assigned with the movement

direction, which is needed for knowing which was the last key pressed by the player and adjust the weapon in the same direction. Additionally lines 144 and 146 of the code adjust the speed for the rigidbody, which is a physics component of the player. Of course only the y variable will get speed, because if the player moves up it is only on the y axis, not the x axis. Then in line 148 the animator component of the player is set to another integer value and as shown before, this has an effect on which animation is played in the animation controller. Additionally line 150 checks if the player has a weapon equipped and when it is true, then it will turn the weapon in the direction of the player movement.

4.3.2 Inventory and Equipment System

Next I will explain the inventory and item system. The inventory system basically consists of an array of game objects and has an `add_item` and a `delete_item` function.

```
30 function add_Item (item : GameObject) {
31
32 var newItems = new Array(Items);
33 newItems.Add(item);
34 Items=newItems.ToBuiltin(GameObject);
35
36 }
```

Figure 12. The `add_item` function of the “Inventory” script.

Figure 12 shows the short `add_item` function, which simply adds a game object to the Item array. The important thing about this function is that first a new array is created with the items of the already existing items in the inventory. Then as the next step the new item is added to the new array and in the last line the inventory array will be recreated with the new item. That means that the arrays change between a dynamically created JavaScript array and a static array. This way makes it possible to easily change the arrays with the JavaScript functionality and then change it back to the static array, which is more efficient.

```
45 for(var i:GameObject in newItems){
46     //if the item has been found
47     if(i == item){
48         //delete item at position
49         item_for_delete=i;
50         newItems.RemoveAt(index);
51         shouldend=true;
52         //break the loop when item was found
53     }
54 index++;
55     if(shouldend){ //Exit the loop
56         Items = newItems.ToBuiltin(GameObject);
57         Destroy(item_for_delete);
58         //update inventory_display
59         playersInvDisplay.remove_content_grid (item.GetComponent(Item).item_content);
60         return;
61         }
62     }
63 }
```

Figure 13. Part of the `delete_item` function of the “Inventory” script.

Figure 13 shows the `delete_item` function, which has been shortened to its more important part. Basically it is done as the `add_item` function just finding and deleting the item instead of adding it. The first step is to search for the item with the for loop in the `newItems` array, which is a dynamical array as in the `add_item` function. Then if we found the item we would use the built-in `RemoveAt` function and remove the item at the `index` location. If the item is found, `shouldend` will be set to true and the next if condition is executed and the `Items` array is again rebuilt with the new content of the dynamic array. Then the item is destroyed and in line 59 the “inventory display” of the player is updated, which means, that the GUI of the inventory is updated and the item is removed from the GUI list. This is the main concept of the “Inventory” script.

Next is the “Item” script, in which all the functionality of the items in the game is located. The “Item” script has several public variables, which determine the type of the item. There are many different types of items in the game such as weapons, seeds, food and equipment, so it is necessary to make them different.

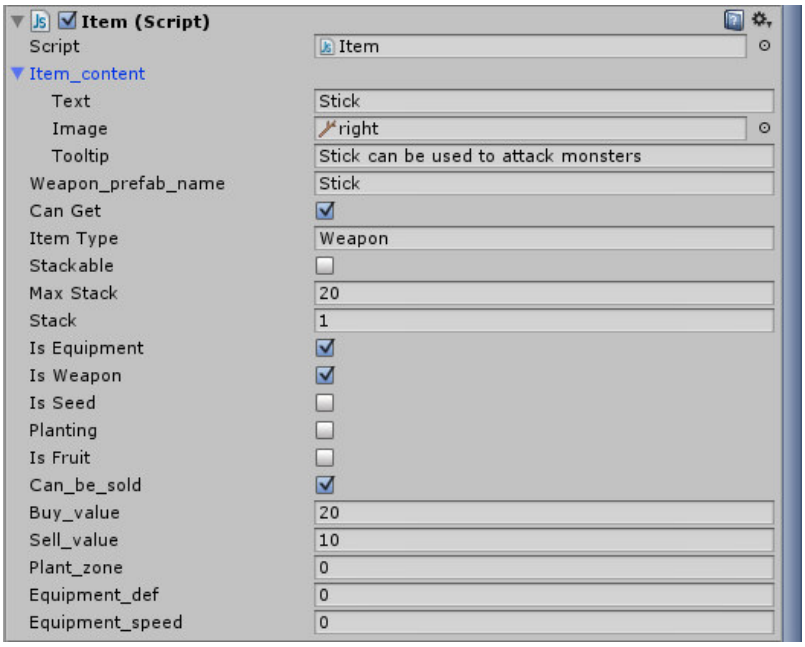

Figure 14. The “Item” script in Unity for item “Stick”.

Figure 14 shows an example item which is a stick. “Item_content” contains the name of the image and the tooltip of the item. It is important later for the “inventory display”, which displays the item sprite with the name and tooltip there. “Weapon_prefab_name” is the prefab name for the weapon which will be created when the item is equipped. Additionally “Can Get” shows if the item can be taken from the floor or not. The “Item Type” informs the “Equipment” script to which slot to put the item, which is important for equipment and weapons. The “stackable” variable means if the item can be stacked and “Max Stack” how many items of the same name can be stacked into one field in the inventory. Moreover “Stack” shows how many stacks the item already has. The following Booleans shows if the item is an equipment, a weapon, a seed or a fruit. The Boolean “Planting” refers to, if the player is close enough to a field where seeds can be planted. Next is the “Can_be_sold” Boolean which shows if the item can be sold or not. The “Buy_value” and “Sell_value” are the gold value for selling and buying and “Plant_zone” refers to where the plant can be planted. Moreover “Equipment_def” and “Equipment_speed” give the bonus defense and bonus speed, which the item adds to the player when equipped.

The most important function for the item script is the `Pickup_item` function, which enables together with the “first_person_pickup” script to pick up items from the ground. The “first_person_pickup” script basically just checks every frame of the game if the distance to the player is smaller than the pickup distance and then if the “e” key is

pressed the `Pickup_item` function of the “Item” script is called and the item is moved to the player’s inventory.

```
function Pickup_item (){
//if the item is stackable
var unique = true;
if (stackable == true){
    var located_it : Item;
    for(var i : GameObject in player_inventory.Items){
        if(i.GetComponent(Transform).name == this.transform.name){
            var j : Item = i.GetComponent(Item);
            if(j.stack < j.maxStack){
                located_it=j;
                }
            }
        }
    //found the same item so will be stacked
    if(located_it!=null){
        unique = false;
        located_it.stack+=1;
        //destroy the item
        Destroy(this.gameObject);
        }
    else{
        unique = true;
        }
    }
//if the item is not yet in inventory put it in and the inventory is not full yet.
if((unique == true)&&(player_inventory.Items.length < player_inventory.size)){
    //adds item to the inventory
    player_inventory.add_Item(this.gameObject);
    //adds item to grid for display
    player_inventory.GetComponent(Inventory_display).add_content_grid(item_content);
    // disable gamecomponents
    if (GetComponent(SpriteRenderer) != null)
    {
        GetComponent(SpriteRenderer).enabled = false;
    }
    GetComponent(Item).enabled = false;
    //its not possible to pick it up again
    canGet =false;
    transform.position.x =0;
    transform.position.y =0;
    }
```

Figure 15. The `Pickup_item` function of the “Item” script.

Figure 15 shows the `Pickup_item` function. At the beginning of the function it will be checked if the item is stackable or not. If it is stackable it will be checked if the item is already in the inventory, so that it can be stacked without creating a new entry in the inventory. If it gets located, the stack variable of the item will be increased by one and the item will be destroyed. Moreover If the item is not yet in the inventory and the inventory is not yet full, it will be added to the inventory and added to the “inventory display” GUI. As well the `SpriteRenderer` is disabled, so that it will not be displayed anymore in the game and the “Item” script will be disabled. Additionally the item will be moved outside of the map.

Another important script related to items is the “Item_Effect” script, which does all the functionality of clicking items in the inventory and equipment window. Moreover it handles all the item effects. It has one important function `UseEffect`, shown in figure 16.

```
38 function UseEffect ()
39 {
40     // if Item is a seed and use check if player stands on a tile where he can plant it
41     if(item.isSeed)
42     {
43         //check if it the player is on a growable field
44         if((Player.GetComponent(Mainchar).planting >0)&& item.plant_zone == Player.GetComponent(Mainchar).plant_area)
45         {
46         //plays sound
47         player_inventory.GetComponent(Inventory_display).play_audio_use_seed ();
48         //FindObject where to plant the seed.
49         var tile = GameObject.FindWithTag("On");
50         var tile_ground = tile.GetComponent(Ground);
51         if(tile_ground.occupied==false)
52         {
53             tile_ground.occupied=true;
54             var seed=   this.gameObject.GetComponent(Seed);
55             var instance : GameObject = Instantiate(Resources.Load("Plants/"+seed.plant_name, GameObject));
56             instance.transform.parent= tile.transform;
57             instance.transform.position.x=tile.transform.position.x;
58             instance.transform.position.y=tile.transform.position.y+0.32;
59             instance.transform.localScale.x=2;
60             instance.transform.localScale.y=2;
61             instance.GetComponent(Plants).set_grownow();
62         //This will delete the item on use or remove 1 from the stack (if stackable).
63         if (deleteOnUse == true)
64         {
65             DeleteUsedItem();
66         }
```

Figure 16. First part of the `UseEffect` function of the “Item_Effect” script.

Figure 16 shows the first part of the `UseEffect` function. This part will test if the item is a seed and if it is true, then it will check if the player is on the planting ground. Then it will play the “planting sound” when the condition is true. Moreover it will check if the planting ground is already occupied by a plant or not. If that is not the case, it will make the ground occupied and then in line 55 an object will be created by loading a prefab from the plants folder and it is given the position and scale and then the plant function `set_grownow` is called, which activates the growing process. This will be explained in detail later in the planting chapter. Additionally the `DeleteUsedItem` function deletes the item when it is used or subtracts the stack if its stack is more than one.

```
71      //if item is eatable
72      if((item.isFruit == true)&&(Player.GetComponent(Mainchar).get_shop() == false)){
73      //plays sound
74      player_inventory.GetComponent(Inventory_display).play_audio_use_food ();
75      //heals player
76      Player.GetComponent(Mainchar).health+= GetComponent(Fruit).heal;
77      //if the hp got over max Health change back to max health
78      if(Player.GetComponent(Mainchar).health > Player.GetComponent(Mainchar).max_health ){
79      Player.GetComponent(Mainchar).health = Player.GetComponent(Mainchar).max_health;
80      }
81      //adjust healthbar
82      GameObject.Find("Playerhealth").GetComponent(texture_follow).update_player_health ();
83      //food to player
84      Player.GetComponent(Mainchar).food+= GetComponent(Fruit).food;
85      //checks if food over max value
86      if(Player.GetComponent(Mainchar).food > Player.GetComponent(Mainchar).max_food ){
87      Player.GetComponent(Mainchar).food = Player.GetComponent(Mainchar).max_food;
88      }
89              if (deleteOnUse == true)
90          {
91              DeleteUsedItem();
92          }
93      }
```

Figure 17. Second part of the `UseEffect` function of the “Item_Effect” script.

Figure 17 shows the second part of the `UseEffect` function, but this time this part is used for eatable items, such as the fruits the player can harvest from the plants, he planted. Moreover it checks as well if the player is not on a shop. Then the game plays an “eating sound”. Additionally the health and food is added from the fruit to the player and in line 82 it adjusts the new player health as well on the player health bar. Moreover the code always checks that the food and health value cannot get over the maximum.

```
95 //checks if item can be sold and Player is at a shop
96      if((item.can_be_sold == true)&&(Player.GetComponent(Mainchar).get_shop() == true)){
97      //sell item
98      Player.GetComponent(Mainchar).gold += item.sell_value;
99      //play audio in invetory_display
100     player_inventory.GetComponent(Inventory_display).play_audio();
101     //message item has been sold
102     player_inventory.GetComponent(Inventory_display).Item_sold = true;
103     player_inventory.GetComponent(Inventory_display).item_sold_content = item.item_content;
104     player_inventory.GetComponent(Inventory_display).item_sold_value = item.sell_value;
105     DeleteUsedItem();
106     }
```

Figure 18. Third part of the `UseEffect` function of the “Item_Effect” script.

Figure 18 shows the third part of the `UseEffect` function, which is about selling items. In this case it will check if the item can be sold and the player is standing on a shop. Then it will sell the item, the player will get the money and it will play a “selling sound”. Additionally the text at the inventory display will change for a short duration and will

display what has been sold and for how much gold. In the end the item will be deleted again.

```
108 //checks if the item can be equipped
109     if((item.isEquipment == true)&&(Player.GetComponent(Mainchar).get_shop() == false)){
110     //equips item
111     equipment.Equipment_on(this.gameObject);
112     }
```

Figure 19. Fourth part of the `UseEffect` function of the “Item_Effect” script.

Figure 19 shows the last part of the `UseEffect` function. This is important for all the equipment which will be put on if clicked with the mouse and when the player is not standing on a shop. The next topic is the equipment system, where the `Equipment_on` function will be explained.

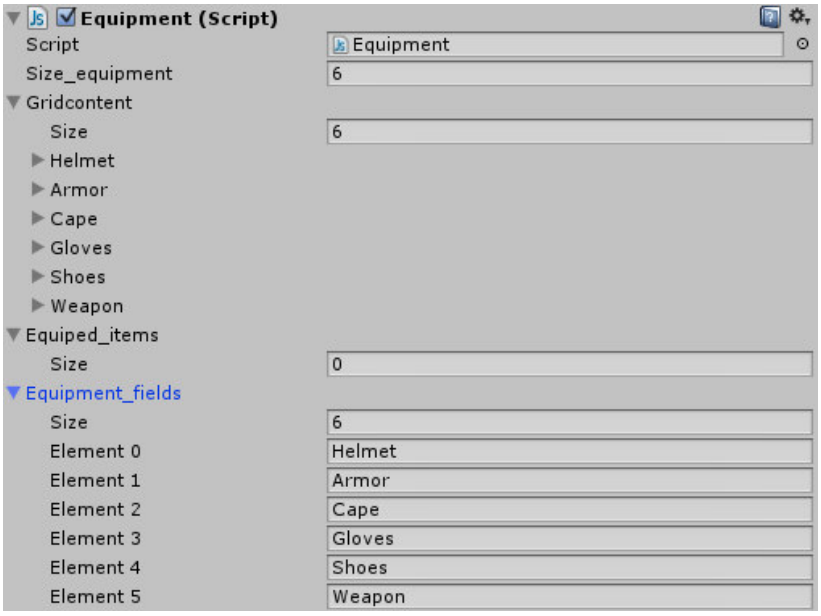

Figure 20. The “Equipment” script in Unity.

Figure 20 shows the most important parts of the public variables for this script. It shows the equipment size is six and the names are all predefined with “Helmet”, “Armor”, “Cape”, “Gloves”, “Shoes” and “Weapon”. These are the six different equipment places in the equipment window. It means the player can wear those six items at the same time and get bonuses from them. The “Gridcontent” is for the GUI window of the equipment, “Equiped_items” are the items which are currently equipped and “Equipment_fields” has fields with type names of the items which can be equipped on each field.

```
118 //for equipping items
119 function Equipment_on (item : GameObject) {
120 //plays sound when Equipment is put on
121     play_audio();
122     for(var i=0; i<size_equipment; i++){
123 //not occupied already and not null
124         if(Equipment_field_occupied[i]==false){
125 //checks if the field is the one the item should get to
126             if(Equipment_fields[i] == item.GetComponent(Item).itemType){
127 //if the equipment is a weapon call place weapon function
128                 if(item.GetComponent(Item).isWeapon == true){
129                     PlaceWeapon(item);
130                 }
131 //adds item to the equipement array
132                 var clone = Instantiate(item, item.GetComponent(Transform).position, item.GetComponent(Transform).rotation);
133                 clone.name = item.name;
134                 Equiped_items[i] = clone;
135 //deletes item from the inventory
136                 player_inventory.delete_Item(item);
137 //changes the field to occupied
138                 Equipment_field_occupied[i] = true;
139 //adds equipment boni def and speed
140                 player.GetComponent(Mainchar).defense += item.GetComponent(Item).equipment_def;
141                 player.GetComponent(Mainchar).speed += item.GetComponent(Item).equipment_speed;
142 //Adds the item content to the grid for displaying it later
143                 Gridcontent[i]= item.GetComponent(Item).item_content;
```

Figure 21. The `Equipment_on` function of the “Equipment” script.

Figure 21 shows the `Equipment_on` function, which handles equipping items, which are equipment. In this function at first the “equipment sound” will be played when the equipment is placed, then the equipment array will be searched and if it is not occupied yet and has the same type as the empty equipment slot, the equipment will be copied and placed in the empty slot. Additionally if the equipment is a weapon, it will call the `Place_weapon` function. Next the item from the inventory will be deleted after the copying to the equipment array and the equipment slot will be marked as occupied, so that not two equipments of same type can be equipped at the same time. Moreover the bonuses for defence and speed will be added to the player and in the end the new equipment will be added to the `Gridcontent` array, which holds the information about the equipment and will be displayed to the player.

```
function Equipment_off (item : GameObject)
{
//plays sound when Equipment is put off
    play_audio();
    for(var i=0; i<size_equipment; i++)
    {
    //checks only for equipped items
        if((Equipment_field_occupied[i]==true))
        {
        //checks if the field is the one the item should be removed from
            if(Equipment_fields[i] == item.GetComponent(Item).itemType)
            {
                //subtracts equipment boni def and speed
                player.GetComponent(Mainchar).defense -= item.GetComponent(Item).equipment_def;
                player.GetComponent(Mainchar).speed -= item.GetComponent(Item).equipment_speed;
                //removes item from equipement array
                Equiped_items[i]= null;
                //changes the field to not occupied
                Equipment_field_occupied[i] = false;
                //Adds the item content to the grid for displaying it later
                Gridcontent[i].image = null;
                Gridcontent[i].text = Equipment_fields[i];
                Gridcontent[i].tooltip = "";
```

Figure 22. The `Equipment_off` function of the Equipment script.

Figure 22 shows the `Equipment_off` function. This does basically the opposite of the `Equipment_on` function. At first it again plays the sound when unequipping the item and then it searches for the equipped items and checks if the item is the same type. When the equipment is found, which should be unequipped then it reduces the bonuses on defence and speed. Additionally the item will be removed from the equipment array with setting it to `null`. Then the `Equipment_field_occupied` array will be set to “not occupied” and in the end the `Gridcontent` will be changed to default for the field, and with that the item has been unequipped.

```
function PlaceWeapon (item : GameObject )
{
    //position of the player
    var vec : Vector3;
    vec.x = player.GetComponent(Transform).position.x;
    vec.y = player.GetComponent(Transform).position.y;
    vec.z = 0;
    //create weapon in hand of player
    var instance : GameObject = Instantiate(Resources.Load("Weapons/"+item.GetComponent(Item).weapon_prefab_name, GameObject),
                                            vec,player.GetComponent(Transform).rotation);
    instance.GetComponent(Weapons).turn_weapon(player.GetComponent(Mainchar).get_last_key(),vec.x,vec.y);
    instance.GetComponent(Weapons).active();
    instance.GetComponent(BoxCollider2D).enabled=false;
    weapon_in_hand = instance;
    player.GetComponent(Mainchar).set_current_weapon(instance);
    player.GetComponent(Mainchar).set_has_weapon (true);
    player.SendMessage("update_attack");
}
//Removes the weapon from the hand of the Player.
function RemoveWeapon ()
{
    Destroy(weapon_in_hand);
    player.GetComponent(Mainchar).set_has_weapon (false);
    player.GetComponent(Mainchar).set_current_weapon(null);
    player.SendMessage("update_attack");
}
```

Figure 23. The `PlaceWeapon` and `RemoveWeapon` functions of the “Equipment” script.

Figure 23 shows the `PlaceWeapon` and `RemoveWeapon` functions. The `PlaceWeapon` function first gets the x and y position of the player and then creates the weapon with the weapon prefab name of the item which has been equipped. Next the new created weapon will be adjusted to the player's direction with the `turn_weapon` function. Additionally this object will be activated and the "BoxCollider2D" will be disabled, which makes it impossible to damage any monsters without attacking them. Then this game object will be stored in the variable `weapon_in_hand` so that it can be later destroyed. Moreover in the "Mainchar" script it calls the `set_current_weapon` function, which sets this weapon as the current weapon which is used for the attack functionality. At last it tells the "Mainchar" script that the player has equipped a weapon. In the `RemoveWeapon` function it just destroys the weapon object and notifies the "Mainchar" script that the player wears no weapon anymore.

4.3.3 Character System

In the next part I will explain the character system for the player with the important code parts in detail.

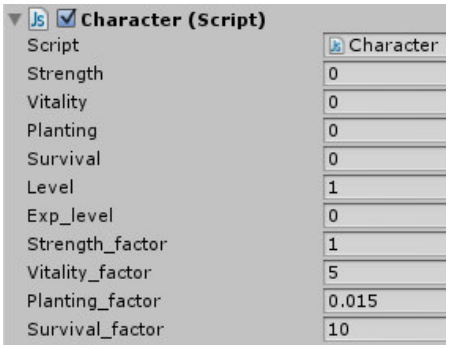

Figure 24.First part of "Character" script in Unity.

Figure 24 shows the first part of the "Character" script in Unity and shows several public variables. The first four variables are the character values, which show how strong the player is in different areas. The player can put points in the different categories when he gains a level. This happens when the player has collected enough experience points by killing monsters. For each level up the player gains five points to distribute between the four different areas. If the player puts points in strength he will do more damage to monsters. If he chooses vitality, he will get more health points and will survive more monster attacks. Additionally the planting attribute will reduce the planting

time for growing crops. The last attribute survival gives the player a higher capacity of food, which makes it easier for the player to survive in the game. Then the other variables in figure 24 show the current level of the player which is one at the start of the game. Then it shows the variable “exp_level” which can be changed to increase the exp which is needed for each level. Then the attribute factors tell how strong effect each point in the attributes has for the player.

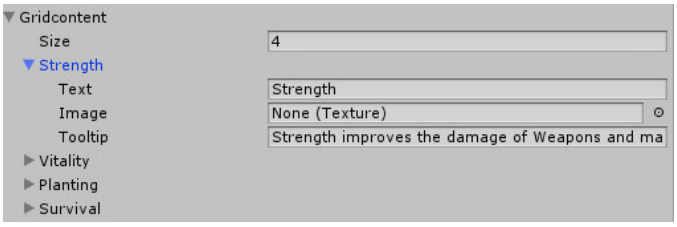

Figure 25. Second part of “Character” script in Unity.

Figure 25 shows the second part of the character script in Unity, which shows the “Gridcontent” which is used for the character display window in the game and shows the different attributes with name and tooltip. The most important function in the character script is the `level_up` function, which checks every frame if the player’s experience is high enough to level up and gives the five stat points for each level he reached. Moreover after reaching a level, the player will need more experience for reaching the next one.

4.3.4 Attack System

The next part is the attack functionality for the player. When the player presses the attack button, the player’s character will attack with its current weapon in the direction the character shows. If the weapon hits a monster, it will get damaged based on the strength of the character and the damage value of the weapon.

```
288     if (Input.GetKeyDown(attack_key)&&(Time.time>next_attack)&&(has_weapon == true))
289     {
290     next_attack = Time.time + Current_weapon.GetComponent(Weapons).speed;
291     play_audio_attack_sword ();
292     attack() ;
293     }
```

Figure 26. The `attack` function call in `Update` function of the “Mainchar” script.

Figure 26 shows where the attack function is called in the `Update` function of the "Mainchar" script. This code will check if the player presses the attack key, if the attack delay is over and if the character has an equipped weapon. Then it plays a sound and calls the `attack` function.

```
431 function attack () {
432 //animation
433 StartCoroutine(animator_time (0.8f));
434 //collider
435 StartCoroutine(weapon_collider_time(0.8f));
436 //stop move player
437 StartCoroutine(stop_move_player_time(0.5f));
438 //plays the weapon animation in the right direction
439 if(last_key == "left"){
440     Current_weapon.GetComponent(Animator).SetTrigger("left");
441 }
```

Figure 27. First part of the `attack` function of the "Mainchar" script.

Figure 27 shows the first part of the attack function which shows the first three function calls, which are "Coroutines", in which it is possible to make the function wait for a time and then continue where they left off. This is important for this kind of functionality. After those three functions the weapon animation will be displayed depending on which direction the player last watched. In figure 27 it is only shown for the left side, but the code is for each direction.

```
477 //put weapon again into right position when attack is finished
478 if(last_key == "left"){
479 Current_weapon.GetComponent(Weapons).turn_weapon("left",transform.position.x-0.33,transform.position.y);
480 }
```

Figure 28. Second part of the `attack` function of the "Mainchar" script.

Figure 28 shows the second part of `attack` function, which shows the code how the weapon is placed back to the normal sprite after the attack animation. Of course it is again done for every direction.

```
494 function animator_time (duration : float){
495 Current_weapon.GetComponent(Animator).enabled = true;
496 yield WaitForSeconds (duration);
497 Current_weapon.GetComponent(Animator).enabled = false;
498 }
```

Figure 29. The `animator_time` function of the "Mainchar" script.

Figure 29 shows the `animator_time` function which enables the animator for the `duration` and then disables it again, so that the animation is stopped after the attack. The `weapon_collider_time` and the `stop_move_player_time` function have almost the same code as this function. In the case of the weapon_collider_time function, the difference is, that the function just activates and disables the “BoxCollider2D”, which makes it possible to hit monsters. Moreover the difference in the `stop_move_player_time` function is, that it changes the `stop_move` variable to true or false, which makes the character unable to move for the duration of the attack.

The next point is how the character can damage the monster. Now it is known how the character can attack but not yet how the damage gets transferred to the monster.

```
21 function OnTriggerEnter2D(coll: Collider2D) {
22     if (coll.gameObject.tag == "Enemy")
23     {
24     coll.gameObject.GetComponent(Monster).damage_monster(Damage+(player.GetComponent(Character).strength *
25     player.GetComponent(Character).strength_factor));
26     }
27 }
```

Figure 30. The `OnTriggerEnter2D` function of the “Weapons” script.

Figure 30 shows the `OnTriggerEnter2D` function, which checks if a “BoxCollider2D” with a trigger enters the area of the weapon and collides. In this case it will collide only if the object it collides with is tagged as “Enemy”, which are the monsters in the game. So if the weapon collides with a monster, it will call the `damage_monster` function and tell the function how much damage the monster receives.

```
109 function damage_monster(Damage : int){
110 var timer1 = Time.time - start_time;
111 if (timer1>1){
112     //play weapon hit sound
113     Player.GetComponent(Mainchar).play_audio_sword_hit ();
114     //display damage on gui
115     got_damaged = true;
116     damage_amount = Damage;
117     health -= Damage;
118     start_time = Time.time;
119     }
120 }
```

Figure 31. The `damage_monster` function of the “Monster” script.

Figure 31 shows the `damage_monster` function, which has at first a timer to make sure the monster is not hit in each frame from the weapon. Then the hit sound is played and `got_damaged` is set to true, which then displays the `damage_amount` on the GUI in the game. Additionally the health of the monster is reduced by the damage sent by the function call.

Next I will explain how the character can be damaged by monsters. This happens when the monster touches the character, or in more detail, when the monster's "BoxCollider2D" collides with the character's "BoxCollider2D".

```
368 function OnCollisionEnter2D(coll: Collision2D) {
369     if (coll.gameObject.tag == "Enemy")
370     {
371         damaged(coll.gameObject.GetComponent(Monster).damage);
372     }
373 }
```

Figure 32. The `OnCollisionEnter2D` function of the "Mainchar" script.

Figure 32 shows the `OnCollisionEnter2D` function, which is called when a game object collides with the character and it is tagged as "Enemy". Then the `damaged` function is called in the "Mainchar" script with the damage of the monster.

```
377 function damaged (dam : int){
378 var timer1 = Time.time - start_time;
379 if (timer1>3){
380     //monster attack sound
381     play_monster_attack ();
382     //call to show damage
383     got_damaged= true;
384     //saves the damage
385     damage_amount = dam - defense;
386     if(damage_amount <= 0){
387         damage_amount =0;
388         //play sound
389         play_no_damage();
390         }
391     else{
392         //play sound
393         play_player_damaged();
394         //low health warning
395         if(health < 0.2* max_health){
396             play_player_almost_dead ();
397             }
398         }
399     health -=  damage_amount;
400     if(health <=0){
401         health = 0;
402         }
403     start_time = Time.time;
404     }
405 }
```

Figure 33. The `damaged` function of the "Mainchar" script.

Figure 33 shows the `damaged` function in the “Mainchar” script, which is called when the character collides with the monster. At first it makes sure the player has not been hit yet for a certain amount of time. Then if the condition is fulfilled, the monster attack sound will be played and the `got_damaged` variable will be set to true to show the damage on the GUI with the damage of `damage_amount`. Additionally if the damage is zero it will play another sound than when the player is hit with damage. Next the health of the player will be reduced by the damage of the monster, which is reduced by the defense of the player, which can be obtained by equipping armor and other items. Moreover if the player’s health is less than 20 percent of the maximum health, then the function `play_player_almost_dead` will be called, which plays a warning sound. If the health of the player is zero, it will be checked in the `update` function of the “Mainchar” script and then will change the game to the “Lost” screen. This is shown in figure 34.

```
297 //Healthcheck
298 if((health<=0)&&(end_game == false)){
299     animator.SetBool("Dead",true);
300     end_game = true;
301     //starts the Lost screen
302     StartCoroutine(dead());
303     }
```

Figure 34. Health check in the `Update` function of the “Mainchar” script.

Figure 34 shows the code which checks each frame if the character’s health is equal to zero or less. Then it sets the animator to the death animation and sets the `end_game` variable to true, which makes sure that this condtion is not entered again. Then the character’s death sound is played and the “Coroutine” `dead` is started. This “Coroutine” just waits for the animation to play and then changes the game scene to the “Lost” screen.

These were the most important aspects of the character with movement, inventory and item system, equipment system, character system, attack functionality, damage functions and player death check. The following section 4.4 is about planting, plants and fruits.

4.4 Planting

In this section all relevant functions and functionality of planting, plants and fruits will be explained. The planting works in the following way. The player has to move on the field where it is possible to plant and then clicks the left mouse button on the seed to plant it. After some time the plant will grow, and when it is ready, it will be possible for the player to harvest it with the press of a button, which is the “e” key.

```
15 function OnTriggerEnter2D(other: Collider2D)
16 {
17     if (other.gameObject.tag == "Player")
18     {
19         other.gameObject.GetComponent(Mainchar).plant_area = zone;
20         other.gameObject.GetComponent(Mainchar).planting +=1;
21         gameObject.tag="On";
22     }
23 }
24
25 function OnTriggerExit2D(other: Collider2D)
26 {
27     if (other.gameObject.tag == "Player")
28     {
29         other.gameObject.GetComponent(Mainchar).planting -= 1;
30         gameObject.tag="Untagged";
31     }
32 }
```

Figure 35. The `OnTriggerEnter2D` function of the “Ground” script.

Figure 35 shows the code in the “Ground” script, which is attached to the tiles which plants can be planted on. The `OnTriggerEnter2D` function is called when a “rigid-body” enters the area of the field. In this case with the condition only the game object tagged with the String player has an effect on the ground field. Moreover when the player enters the area of the field, it will add one to the `planting` variable in the “Mainchar” script, which notifies it that the player is on the growable field. Additionally the code changes the `plant_area` variable into the zone in which the field is. This notifies the “Mainchar” script which plants can be planted on the field. The last line tags the field with the String “on”, which marks the field which the plant should be planted on. The other function `OnTriggerExit2D` is the opposite of the function before. It is called when the player leaves the area of the field and then the `planting` variable in the “Mainchar” script is subtracted by one and the tag on the field is removed.

Next when the player clicks with the left mouse button on the seed in the inventory UI, it will call the `UseEffect` function. Most parts of this function were explained before in section 4.3.2, however the `set_grownow` function and how the plants are growing were not explained yet and will be explained next.

```
function Update () {
    if(grow_now == true){
        if(improved_grow == true){
        grow_time = grow_time - Mathf.Round(player.GetComponent(Character).planting_factor *
        player.GetComponent(Character).planting * grow_time);
        improved_grow = false;
        }
    grow();
    }
    if (Input.GetKeyDown(ButtonToPress)&& (grow_ready==2)&& (transform.parent.tag=="On")){
        create_fruit();
    }
}
```

Figure 36. The `Update` function of the “Plants” script.

Figure 36 shows the `Update` function of the “Plants” script. This function is called every frame of the game and checks if the `grow_now` variable is true, which becomes true when the `set_grownow` function is called, as mentioned in section 4.3.2. The `improved_grow` condition will change the growing time of the plant, if the player has improved his planting attribute, which he can do when he levels up. Then the `grow` function is called, which can be seen in figure 37.

```
function grow(){
var timer1 = Time.time - start_time;

    if((timer1>grow_time/2) && (grow_ready==0)){
        //change sprite
        GetComponent(SpriteRenderer).sprite = sprite2;
        grow_ready+=1;
    }
    if((timer1>grow_time)&&(grow_ready==1)){
        //change sprite
        GetComponent(SpriteRenderer).sprite = sprite3;
        grow_ready+=1;
    }
    if (grow_ready ==2){
        grow_now =false;
    }
}
```

Figure 37. The `grow` function of the “Plants” script.

Figure 37 shows the `grow` function of the “Plants” script, which makes the plant grow. At first a timer is needed to make the plant grow in a certain amount of seconds. The first condition checks if half of the growing time of the plant is reached. When it is still in the first phase, it will change the plants sprite to the second sprite and the `grow_ready` variable is incremented, which tells in which growing phase the plants are. Additionally that makes sure that the growing proceeds step by step without jumping over one of the growing phases. Furthermore the second condition is almost the same as the condition before, just that it will replace the sprite of the plant with the third

sprite when the whole growing time is over and if the first phase had been completed. In the end, the last condition makes sure that the plant is not growing again after it has already grown.

Then back at the `Update` function in figure 36, the `create_fruit` function is called when the player is on the field with the ready grown plant and the “e” key is pressed on the keyboard.

```
71 function create_fruit () {
72 //creates the fruit object
73 var instance : GameObject = Instantiate(Resources.Load("Plants/"+fruit_name, GameObject));
74     instance.transform.position.x=transform.position.x;
75     instance.transform.position.y=transform.position.y-0.32;
76     instance.transform.localScale.x = 0.7;
77     instance.transform.localScale.y = 0.7;
78     instance.name = fruit_name;
79     //destroys the plant
80     transform.parent.GetComponent(Ground).occupied = false;
81     grow_ready =0;
82     Destroy (gameObject);
83 }
```

Figure 38. The `create_fruit` function of the “Plants” script.

Figure 38 shows the `create_fruit` function of the “Plants” script with which the fruit of the plant is created. At first it creates a clone of the fruit prefab with the fruit name of the plant and then adjusts the location and scale of the new created object. When everything is done the plant will be destroyed and only the fruit is on the ground where the plant was before. These are the basics of the planting mechanics. Monsters will be discussed in section 4.5.

4.5 Monsters

4.5.1 Basics

In this section I will explain the most important functionalities of the monsters in the game. There are three different types of enemies: hunters, plant eaters and bosses. The hunters are only interested in the player and try to kill him. Additionally the plant eaters are focusing on eating plants and have no interest in the player himself. The bosses are bigger and stronger and usually protect the gate to another area and only try to kill the player if he attracts their attention. At first I will show the animations with the Unity built-in Mecanim tool.

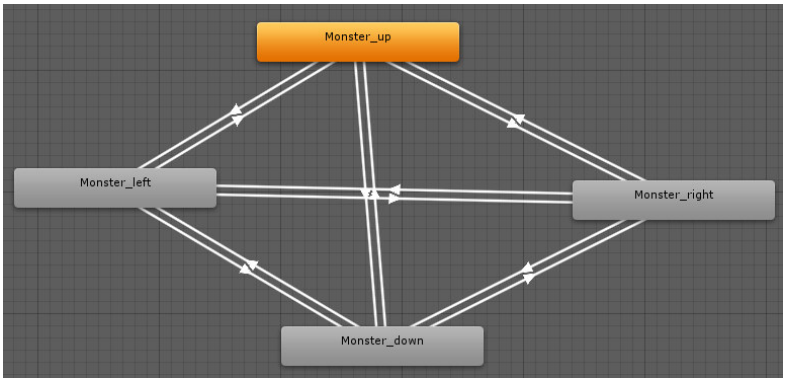

Figure 39. The Mecanim animation tool from Unity: Monster movement.

Figure 39 shows the animations for the monsters. Each monster has four different animations in each direction, which can change between each other. So it is almost the same as for the player movement, which has been shown in section 4.3.1.

```
66 function Update () {
67     if(health <=0){
68         health = 0;
69         //give gold to player
70         Player.GetComponent(Mainchar).gold += gold;
71         //give exp to player
72         Player.GetComponent(Mainchar).exp += exp;
73         //checks if monster is a boss then change back music to normal
74         if((gameObject.name.Contains("boss") == true)&&(once_change_music == false)){
75             GameObject.Find("Game").GetComponent(Music).play_music(GameObject.Find("Game").
76             GetComponent(Music).current_music);
77             once_change_music = true;
78             //open gate again which has been closed in the boss fight
79             if((gameObject.name == "Eye boss")&&(GameObject.Find("Plant boss") == null)){
80             GameObject.Find("Gate2").GetComponent(BoxCollider2D).enabled = false;
81             GameObject.Find("Gate2").GetComponent(MeshRenderer).enabled = false;
82             }
83             if((gameObject.name == "Plant boss")&&(GameObject.Find("Eye boss") == null)){
84             GameObject.Find("Gate1").GetComponent(BoxCollider2D).enabled = false;
85             GameObject.Find("Gate1").GetComponent(MeshRenderer).enabled = false;
86             }
87             if(gameObject.name == "Gargoyle boss"){
88             GameObject.Find("Gate2").GetComponent(BoxCollider2D).enabled = false;
89             GameObject.Find("Gate2").GetComponent(MeshRenderer).enabled = false;
90             }
91             if(gameObject.name == "Dragon boss"){
92             GameObject.Find("Gate3").GetComponent(BoxCollider2D).enabled = false;
93             GameObject.Find("Gate3").GetComponent(MeshRenderer).enabled = false;
94             }
95             if(gameObject.name == "Shadow boss"){
96             GameObject.Find("Gate4").GetComponent(BoxCollider2D).enabled = false;
97             GameObject.Find("Gate4").GetComponent(MeshRenderer).enabled = false;
98             }
99         }
100        //play monster dead sound and destroys object after playing sound
101        Player.GetComponent(Mainchar).play_monster_dead (monster_dead);
102        Destroy (gameObject);
103    }
104 }
```

Figure 40. The `Update` function of the “Monsters” script.

Figure 40 shows the `Update` function of the “Monsters” script, which is the essential part of the script. In the first part of the code it is checked, if the monster’s health is less than or equal to zero. If the condition is fulfilled then the health is set to zero and the gold variable and the exp variable in the “Mainchar” script are changed and increased by the monster’s value. Moreover the next part of the code is about changing the music back to normal if the monster is a boss and is dead. So the condition is that if the monster’s name contains the String “boss” and the Boolean `once_change_music` is false, then the code will find the game object with string “Game” and will call the `play_music` function from this object with the variable which should be played. Additionally it will change the Boolean variable to true so that the music will be changed only once. The next five conditions are opening the gate again, which has been closed at the boss fight. For this the condition checks the name of the boss to be sure which gate has to be reopened. The last part of the code calls the `play_monster_dead` function from the “Mainchar” script, which plays a sound when the monster dies and then destroys the monster.

4.5.2 Path Finding

Another important part of the monsters is the path-finding system, which is done with Navmesh2D. This tool is used combined with 2Dtoolkit to mark the areas which can be walked on and which are blocked for the monsters. So basically I create two tilemaps on each other, one of which is the visible map and the other one is for the navigation of the monsters.

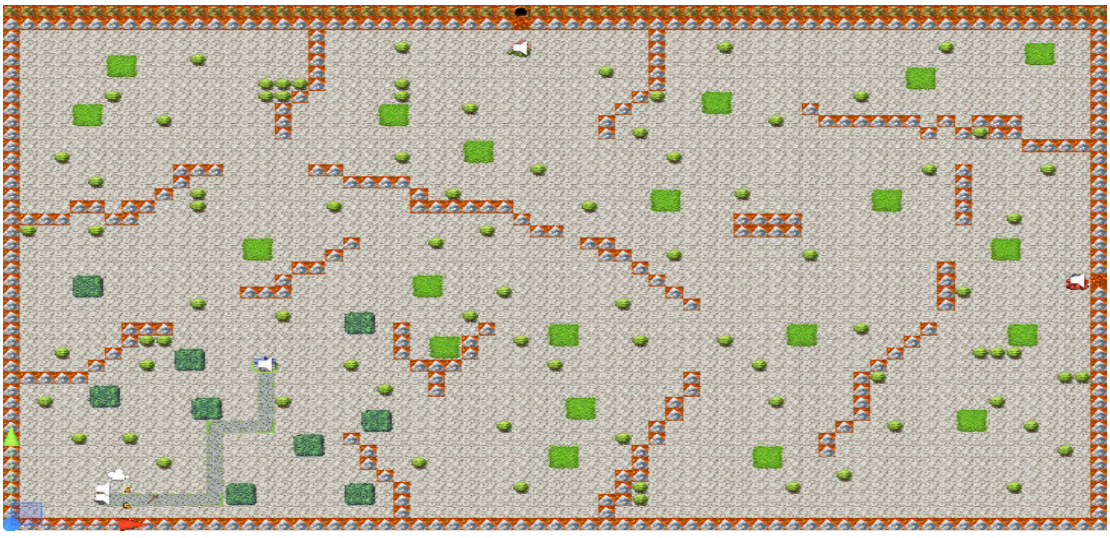

Figure 41. Second tilemap for path-finding.

Figure 41 shows the second tilemap, which contains red for a blocked area, which cannot be walked on and a grey area, which can be walked on. Each of the red objects have the layer “Blocked” and each of the grey objects the layer “Walkable”. This is important for the next step with Navmesh2D.

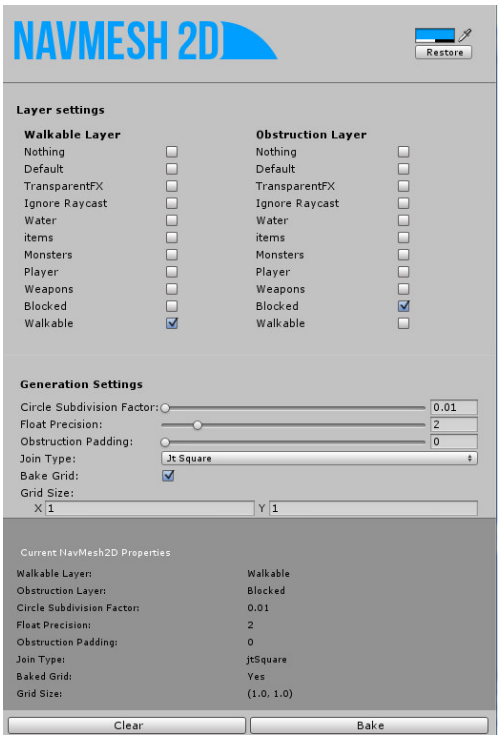

Figure 42. Navmesh2D options in Unity.

Figure 42 shows the Navmesh2D options in the Unity editor. The first part shows the layer settings, which shows all the Unity default layers and all manually added layers. Then it is necessary to mark the layer which can be walked on and the layer which is blocked. In this case it is “Walkable” for the “Walkable” layer and “Blocked” for the “Obstruction” layer. Moreover “Generation Settings” are used to make the “navmesh” more precise for the developer’s needs. The “Circle Subdivision Factor” is the multiplier for subdividing circle colliders and the higher the value the less is the subdivision. Additionally the “Float Precision” defines how many decimal places the float operations are calculated with. It needs to be the higher the smaller the colliders are, to make sure the “navmesh” will be created correctly. Moreover the “Obstruction Padding” defines how big the padding is around the colliders. In this case it is zero to be exactly as big as the colliders are. Next is the “Join Type” which defines how the collider edges are. There are miter, square and round to choose from but in this game the square type is used for better performance. “Bake Grid” enables to bake the navmesh, which means it is saved

in a data file which then does not need to be calculated again and the grid size is the size of the tiles in the map. With the current navmash2D properties is a short summary for the used options and in the end it is necessary to click the bake button to start baking the navmesh. This process can take a long time, if the map has a big area, as in this project where it takes arround 30 minutes to finish baking.

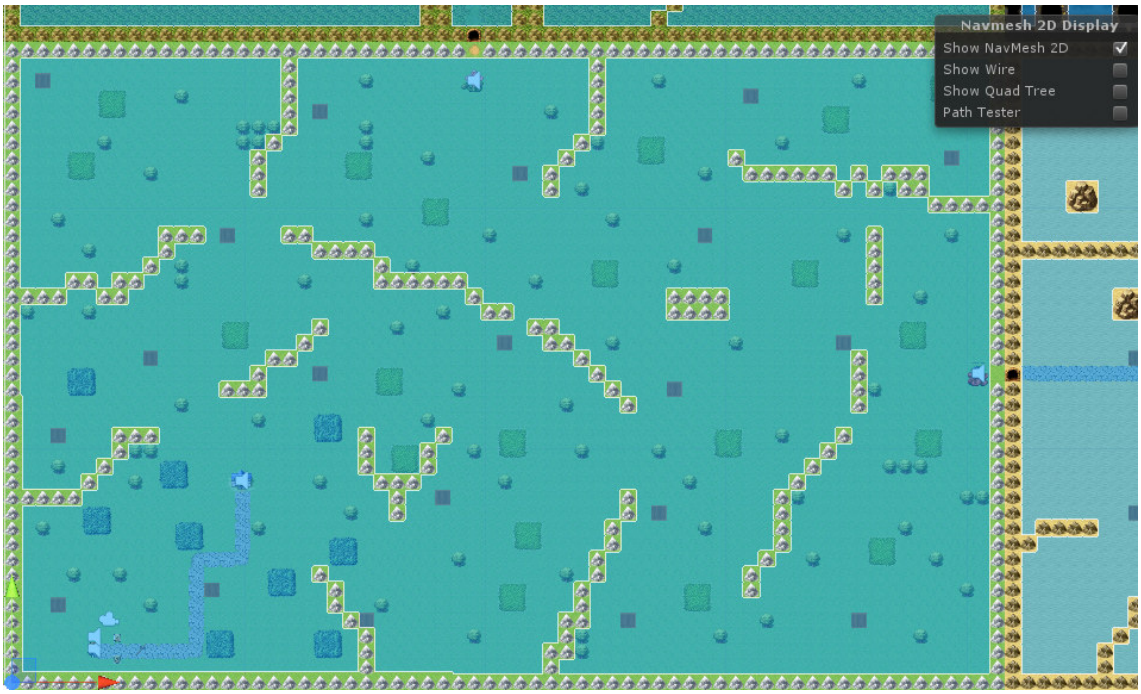

Figure 43. Navmesh2D in map.

Figure 43 shows the ready baked “navmesh” in the map, in which the marked area is the area in which the monsters can move and the mountains is the area where monsters cannot walk on. This limits monsters from entering another area, which would be a problem for the balance of the game. Above I explained how to create the navigation path. Next I will explain how the path-finding works with the scripts.

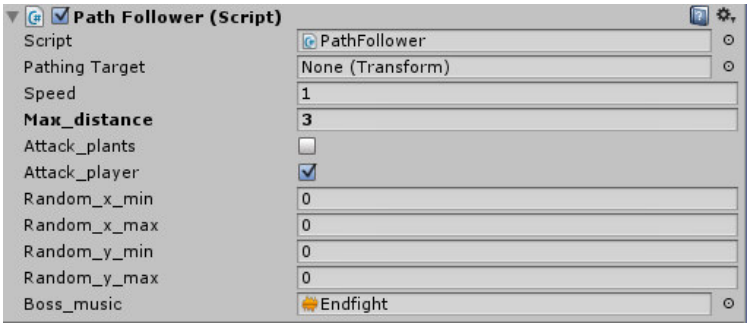

Figure 44. The “Path Follower” script in Unity editor.

Figure 44 shows the “Path Follower” script in the Unity editor with the attributes. The “Pathing Target” is the goal of the monster to move to, which is the most essential part of the variables. The default value is none because not all monsters have a target from the start of the game. Additionally the speed value defines how fast the monster moves and the “Max_distance” value is the maximal distance between the target and the monster until the monster still follows the target. The next two attributes are Boolean values which define if the monster attacks the player or plants. Moreover the random x and y values specify the area in which the monster moves randomly. The following script is in C# code.

```
41     public void Start(){
42         if(attack_player == true){
43         pathingTarget = GameObject.FindWithTag("Player").GetComponent<Transform>();
44         }
45         if(max_distance == 0){
46         max_distance = 5;
47         }
48     }
```

Figure 45. The `Start` function of the “Path Follower” script.

Figure 45 shows the `Start` function of the “Path Follower” script. In this script it checks if the monster attacks the player or not. If the condition is fulfilled then the target for the monster will be the player. Then the next lines of code are defining the maximal distance between the monster and the target, if it has not been set yet.

```
51     void Update () {
52         //if attack plants is true searches for plants
53         if(attack_plants == true){
54             plants = GameObject.FindGameObjectsWithTag("Plant");
55             if(plants.Length!=0){
56                 for(int i=0; i<plants.Length; i++ ){
57                     //find closest plant and make it the target
58                     float distance = Vector2.Distance(plants[i].GetComponent<Transform>().position,
59                                                       transform.position);
60                     if(closest_distance > distance){
61                         closest_distance = distance;
62                         position_in_list = i;
63                     }
64                 }
65                 pathingTarget = plants[position_in_list].GetComponent<Transform>();
66                 closest_distance= 9999F;
67             }
68         }
```

Figure 46. First part of the `Update` function of the “Path Follower” script.

Figure 46 shows the first part of the `Update` function of the “Path Follower” script. In this part it shows the functionality of the monsters which attack plants and not the player. So in the first line of code it checks if the monster attacks plants and if that is

true, it will find all game objects in the game tagged “Plant” and saves them into the `plants` array. Additionally if one or more plants have been found, it will calculate all distances between the monster and all found plants and then when the closest plant is found, it will set this plant object as a target.

```
function OnTriggerEnter2D(other: Collider2D){
    if ((other.gameObject.tag == "Enemy")&&(other.gameObject.GetComponent(Monster).attack_plants==true)){
        //play sound
        other.gameObject.GetComponent(Monster).play_monster_eat_plant();
        //set the field free that it an be planted on again
        transform.parent.GetComponent(Ground).occupied = false;
        //destroy plant if monster can destroy plants
        Destroy (gameObject);
    }
}
```

Figure 47. `The OnTriggerEnter2D` function of the “Plants” script.

Figure 47 shows the `OnTriggerEnter2D` function, which is called when a game object tagged as “Enemy” and if the Boolean of the “Monster” script `attack_plants` is true. So when monsters which have plants as targets collide with the plant, a “monster eating sound” will be played in the “Monster” script. The ground where the plant is growing will be set again to unoccupied so that new plants can be planted on this field again. In the end the plant will be destroyed.

```
        if(pathingTarget!=null){

            float distance = Vector2.Distance(pathingTarget.position, transform.position);

            if(max_distance > distance){
                //final boss changes to Boss component
                if(gameObject.name.Contains("boss")== true){
                        gameObject.AddComponent<Boss>();
                        gameObject.GetComponent<Boss>().boss_music = boss_music;
                        gameObject.GetComponent<Boss>().speed =1;
                        gameObject.GetComponent<Boss>().max_distance = 50;
                        gameObject.GetComponent<Boss>().gate = GameObject.Find("Gate5");
                        gameObject.GetComponent<PathFollower>().enabled = false;
                }
                if (once_path == false) {
                    path = NavMesh2D.GetSmoothedPath(transform.position, pathingTarget.position);
                    once_path = true;
                    StartCoroutine( path_delay ());
        }
    }
```

Figure 48. Second part of the `Update` function of the “Path Follower” script.

Figure 48 shows the second part of the `Update` function of the “Path Follower” script. This part of the script shows what happens when the monster has a target. If this is

fulfilled, then the distance between the monster and the target will be calculated and then another condition will check if the distance is in the maximal distance of the monster. When this is true there will be another condition check. If the name of the monster contains boss, which means it is the final boss monster and so it gets the script boss to the game object and defines the boss music, speed maximal distance, and defines the gate, which will be closed and disables the “Path Follower” script. If the monster is not the final boss monster, then it will check the `once_path` Boolean, which defines if the monster is already following a path, because there can be only one path at a time the monster follows. Moreover if the monster does not have a path yet, then it will get a calculated path over the Navmesh2D function `GetSmoothedPath`, which gets as values the monster and the target’s position. Additionally the `once_path` variable will be set to true and then the “Coroutine” `path_delay` will be called, which sets the `once_path` variable again to false after a short time. This is done because otherwise if the player moves, the monster will walk the whole way where the player was standing before. In this case the monster would never catch the player. So this code always calculates a new path for the monster in a short time gap to make the monsters react to the movement of the player.

```
95          else{
96              if (once_path == false){
97                  Vector3 random_position = new Vector3(Random.Range(transform.position.x+random_x_min,
98                                                                     transform.position.x+random_x_max),
99                                                        Random.Range(transform.position.y+random_y_min,
100                                                                    transform.position.y+random_y_max),0);
101                 path = NavMesh2D.GetSmoothedPath(transform.position, random_position);
102                 once_path = true;
103             }
104         }
105     }
```

Figure 49. Third part of the `Update` function of the “Path Follower” script.

Figure 49 shows the third part of the `Update` function of the “Path Follower” script. This part is activated when the target is not in range of the monster and if the monster has no path yet. Additionally it calculates a random position for the monster to walk to. Furthermore if the monster has no target, it will move randomly as shown in the code.

```
114             if(path != null && path.Count != 0){
115             transform.position = Vector2.MoveTowards(transform.position, path[0], speed*Time.deltaTime);
116             //top
117             if(transform.position.y < path[0].y){
118                 GetComponent<Animator>().SetInteger("Direction", 2);
119             }
120             //down
121             if(transform.position.y > path[0].y){
122                 GetComponent<Animator>().SetInteger("Direction", 0);
123             }
124             //left
125             if(transform.position.x > path[0].x){
126                 GetComponent<Animator>().SetInteger("Direction", 1);
127             }
128             //right
129             if(transform.position.x < path[0].x){
130                 GetComponent<Animator>().SetInteger("Direction", 3);
131             }
132             if(Vector2.Distance(transform.position,path[0]) < 0.01f)
133             {
134                 path.RemoveAt(0);
135             }
136         }
137         else{
138             once_path = false;
139         }
140         }
```

Figure 50. Last part of the `Update` function of the “Path Follower” script.

Figure 50 shows the last part of the `Update` function of the “Path Follower” script. This part contains the movement and animation changes. At first it checks if the monster has a path and the path has at least one node. Next it changes the position of the monster and lets it move towards the next node of the path with the speed multiplied with `deltaTime`, which moves depending on time not frames. Moreover the next four conditions are changing the monsters animation depending on the direction it moves. The last if condition checks if the distance from the monster to the node of the path is smaller than 0.01 “Float” and then removes this node of the path. The “else” statement in the end changes the `once_path` variable to false, if the end of the path is reached or no path is assigned to the monster.

Another script which is important for boss monsters is the “Boss” script. This script is almost same as the “Path Follower” script but it changes the maximal distance of the boss monster when the player once comes too close to it and then the boss monster follows until it or the player dies.

```
49              if(max_distance > distance){
50                  if(once == false){
51                  play_boss_music ();
52                      once = true;
53                      //closes the last gate for the boss
54                      GameObject.Find(gate.name).GetComponent<BoxCollider2D>().enabled = true;
55                      GameObject.Find(gate.name).GetComponent<MeshRenderer>().enabled = true;
56                  }
57                  //if once in range boss follows player everywhere
58                  max_distance = 50;
```

Figure 51. Changes from “Path follower” script in the “Boss” script.

Figure 51 shows the changes in the “Boss” script from the “Path follower” script. The changes are that if the player is in the range of the boss monster, it starts the boss music once. Moreover it activates the gate defined in the Unity editor, which makes sure that the player cannot leave this zone before defeating the boss. Additionally it changes the maximal distance to 50 which is the whole area, so the player has to defeat the monster. If the player kills the monster the old gate will reopen, which can be seen in figure 40 and a new gate will open to the next area, which is shown next in the “Gate” script.

```
11 function Update () {
12     if((GameObject.Find(boss.name) == null )&&(once == false)){
13         GetComponent(BoxCollider2D).enabled = false;
14         GetComponent(MeshRenderer).enabled = false;
15         once = true;
16         }
17 }
```

Figure 52. The `Update` function of the “Gate” script.

Figure 52 shows the `Update` function of the “Gate” script. The gate object is related to one boss in the `boss` variable. So if the `boss` is equal to null, which means it is dead, the “Boxcollider2D” of the gate and the “MeshRenderer”, which draws the gate graphics, are disabled. This results in an open path for the player to walk through to the next area.

4.5.3 Health Bar System

Another important script is the “texture follower” script, which is necessary for the health bar showing up over the monsters and the player.

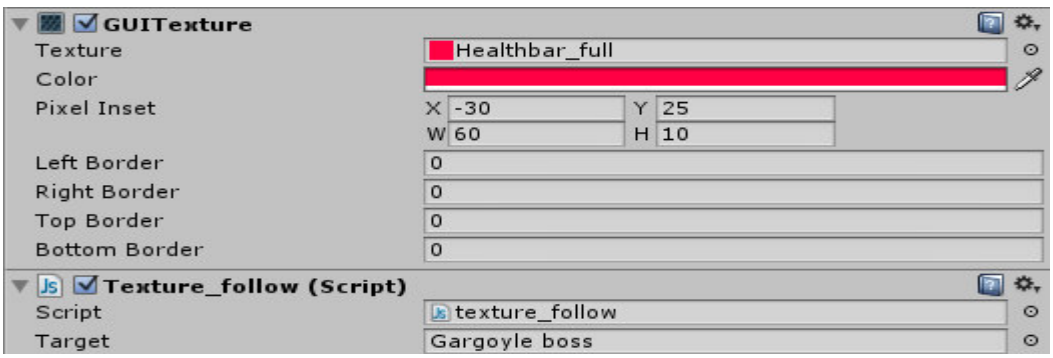

Figure 53. Health bar of monster in Unity editor.

Figure 53 shows the health bar object in the Unity editor. The “GUITexture” has the graphic “Healthbar_full” as texture, which is just a red rectangle. The “Color” attribute is red and the “Pixel Inset” defines the position with x and y and the width and height of the rectangle. The four border attributes define the pixels which are not affected by the scale. Additionally the “Texture_follow” script has the “Target” variable which is the target, the texture is following.

```
30 function Update () {
31 //follows the target
32 wantedPos = Camera.main.WorldToViewportPoint (target.GetComponent(Transform).position);
33 transform.position = wantedPos;
34 }
```

Figure 54. The `Update` function of the “Texture_follow” script.

Figure 54 shows the `Update` function of the “Texture_follow” script. In this function the variable `wantedPos` is assigned with the translated position of the target. This has to be done because Unity uses three different coordinate systems, which are world point system, screen point system and view-port system. So it is sometimes necessary to change the location to a different coordinate system. In the end the health bar position is put to the position of the monster.

```
38     if(monster == true){
39     //loose hp on health_bar of monsters
40     if((target.GetComponent(Monster).got_damaged == true)){
41         GetComponent(GUITexture).guiTexture.pixelInset.width =
42         target.GetComponent(Monster).health/target.GetComponent(Monster).max_health * original_width;
43     }
44 }
45     else{
46         //loose hp on health_bar of player
47         if(target.GetComponent(Mainchar).got_damaged == true){
48             GetComponent(GUITexture).guiTexture.pixelInset.width =
49             target.GetComponent(Mainchar).health/target.GetComponent(Mainchar).max_health * original_width;
50             }
51         }
52 }
```

Figure 55. Part of the `OnGUI` function of the “Texture_follow” script.

Figure 55 shows the important part of the `OnGUI` function of the “Texture_follow” script. This code checks at the beginning if this script is a component of a monster or of the player. Then if it is a monster and the variable `got_damaged` in the Monster script is true, then the width of the GUI texture is changed to the percent, which is calculated by the current health divided by the maximum health of the monster. Moreover if it is not a monster but the player, then it is done the same way. If the “Mainchar” script tells the player got damaged and then the health bar is adjusted.

4.5.4 Spawner

The last part of the monster section is about the monster spawner. This game object with the “Spawn_monster” script spawns the defined types of monsters and number. Each area in the game has one monster spawner, which makes sure that there are always the same number of monsters in the same area and creates the different kind of monsters of the area randomly.

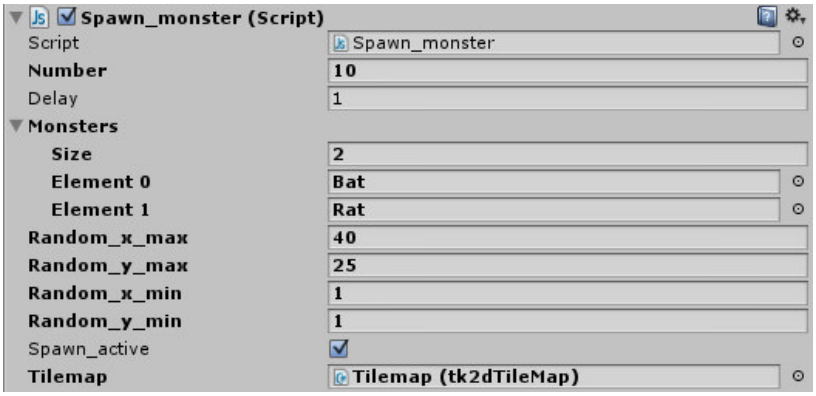

Figure 56. The “Spawn_monster” script in the Unity editor.

Figure 56 shows the variables of the “Spawn_monster” script. The “Number” variable refers to the number of monsters which should be spawned and the “Delay” variable is the delay in seconds between creating the monsters. Moreover the “Monsters” array is adjusted to size two with two different monsters in this case it is the “Bat” and the “Rat”, which are in the starting area. The next four values are for the random position generator, which define the minimum and maximum, where the monsters can spawn. Additionally the “Spawn_active” variable activates to spawn the monsters and if it is disabled, then deletes all monsters from this spawner.

```
29 function Update () {
30     if(count < number){
31         spawn (count);
32     }
33     if(spawn_active == true){
34         for(var i=0; i< Monster_list.length; i++){
35             if(Monster_list[i]==null){
36                 spawn (i);
37             }
38         }
39     }
40 }
```

Figure 57. The `Update` function of the “Spawn_monster” script.

Figure 57 shows the `Update` function of the “Spawn_monster” script. In this code the first condition checks if the number is not reached yet and then spawns monsters if the maximal number of monsters has not been reached yet. The second part of the code checks first with the condition if the spawner is active and if this is fulfilled, then the for loop is executed. This checks if one element of the monster list is equal to null, which means the monster is dead. Additionally when this is the case, then a new monster will be spawned at the place in the array where the monster died.

```
43 function spawn (array_pos : int) {
44 var timer1 = Time.time - start_time;
45     if (timer1 > delay){
46         //choose a random monster from the list
47         var random : int = Random.Range(0, Monsters.Length);
48         var random_x : float = Random.Range(random_x_min,random_x_max);
49         var random_y : float = Random.Range(random_y_min,random_y_max);
50         var vec : Vector3;
51         vec.x = random_x;
52         vec.y = random_y;
53         vec.z = 0;
54         //check if random value is occupied.
55         var tile_id : int = tilemap.GetTileIdAtPosition (vec,1);
56         if(tile_id == -1){
57             var new_monster : GameObject = Instantiate(Monsters[random]);
58             new_monster.transform.position.x = random_x;
59             new_monster.transform.position.y = random_y;
60             Monster_list[array_pos]= new_monster;
61             count++;
62             start_time = Time.time;
63         }
64     }
65 }
```

Figure 58. The `Spawn` function of the “Spawn_monster” script.

Figure 58 shows the `spawn` function of the “Spawn_monster” script. In this function at first the timer is initialized and then in the condition is checked if the “timer1” variable is bigger than the “delay” variable. Then a random number between zero and the size of the monsters array is created, which defines the type of monster for example rat or bat as seen in figure 56. Additionally a random x and y value are created and then it is checked if the random position is inside the tilemap and that it is not on any tile with a top layer tile, which means a blocked tile. If this is the case, a new monster is created with the monster prefab and the random x and y positions. Moreover the new created monster is inserted into the Monster array at the position defined at the call of the function. Furthermore the count variable is incremented by one and the `start_time` is adjusted to the current time.

At any time of the game there is only one spawner active at the same time. This is done because of performance reasons. So in the map there is a trigger at each gate to another zone, which changes the spawning of the monsters. This is done with the “Spawn_activate” script, which is explained next.

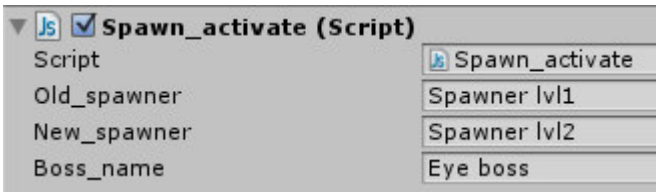

Figure 59. The “Spawn_activate” script in the Unity editor.

Figure 59 shows the public variables of the “Spawn_activate” script. It has the “Old_spawner” variable, which is the spawner from the old area and the “New_spawner” variable, which is the spawner from the new area. Additionally it has the “Boss_name” as variable which is the boss, and which is between those two areas. In the `Update` function of the “Spawn_activate” script it is ensured that this spawn activator is only active when the boss is already dead.

```
17 function OnTriggerEnter2D(coll: Collider2D) {
18
19     if ((coll.gameObject.tag == "Player")&&(once == true))
20     {
21         if(GameObject.Find(old_spawner.name).GetComponent(Spawn_monster).enabled == true){
22         GameObject.Find(old_spawner.name).GetComponent(Spawn_monster).delete();
23         GameObject.Find(new_spawner.name).GetComponent(Spawn_monster).enabled = true;
24         GameObject.Find(new_spawner.name).GetComponent(Spawn_monster).spawn_active = true;
25         }
26         else if(GameObject.Find(new_spawner.name).GetComponent(Spawn_monster).enabled == true){
27         GameObject.Find(new_spawner.name).GetComponent(Spawn_monster).delete();
28         GameObject.Find(old_spawner.name).GetComponent(Spawn_monster).enabled = true;
29         GameObject.Find(old_spawner.name).GetComponent(Spawn_monster).spawn_active = true;
30         }
31     }
32 }
```

Figure 60. The `OnTriggerEnter2D` function of the “Spawn_activate” script.

Figure 60 shows the `OnTriggerEnter2D` function in the “Spawn_activate” script, which is called when the player is in the area between the two zones and the boss is already dead. Then it checks which of the spawners is active, the old one or the new one. Next it calls the `delete` function, which deletes the monster list from the “Spawn_monster” script of the active spawner and then activates the other spawner which was not active before. This is a toggle function, so every time the player goes from one zone to another, it changes the active spawner and there will be only mon-

sters in the same area as the player. The only exception is of course the bosses which are not spawning.

4.6 Audio

This section shows how the sound files and music files are implemented in the game. The music and sound files are all in the OGG compressed format, which ensures the file size is not too big. For playing sounds and music in Unity it is necessary to have an “Audio Source” component at the game object which should play the audio file.

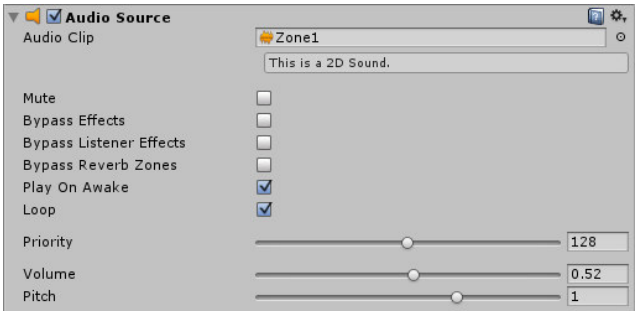

Figure 61. “Audio Source” in the Unity editor.

Figure 61 shows the “Audio Source” component of the “Game” game object. The “Audio Clip” is the audio file, which can be started to play in a script or directly with the component if the “Play On Awake” variable is true. Moreover it can be chosen that this audio file will loop endlessy with the variable “Loop”. Here it is also possible to define the volume and the pitch of the sound. In this case this “Audio Source” will start to play the “Zone1” music when the game is started.

```
3 public var current_music : AudioClip;
4
5 function play_music (new_music : AudioClip) {
6 var new_source : AudioSource = GetComponent(AudioSource);
7 GetComponent(AudioSource).clip = new_music;
8 current_music = new_music;
9 new_source.loop = true;
10 new_source.Play();
11 }
```

Figure 62. The `Play_music` function of the “Music” script.

Figure 62 shows the `Play_music` function in the “Music” script. This function is called for changing the music of the “Audio Source” of the game object, for example the

“Game” object, which plays the background music. With this script the music can easily be changed to another music file. In the code the audio clip of the “Audio Source” is changed to the new music and so replaces the current music with the new one. Then it makes the sound file loop and the `Play` function starts to play the music file from the “Audio Source”. Next will be shown how the game music is changed for the different zones.

```
function OnTriggerEnter2D(coll: Collider2D) {

    if (coll.gameObject.tag == "Player")
    {
        //makes sure the same music is not yet playing and no boss music
        if((GameObject.Find("Game").GetComponent(Music).current_music != new_music)&&
        (GameObject.Find("Game").GetComponent(Music).current_music != boss_music)){
        GameObject.Find("Game").GetComponent(Music).play_music(new_music);
        }
}
}
```

Figure 63. The `OnTriggerEnter2D` function of the “Music_change” script.

Figure 63 shows the `OnTriggerEnter2D` function in the “Music_change” script. The music change script is connected with an area located between two zones. So if the player steps into the area, it will call this function. At first the code checks if the music which is playing at the moment is not the same as the new music and that it is not the boss music. The boss music always plays when the player fights a boss monster. If those two conditions are met, then the new music will be played. User Interface will be explained in section 4.7.

4.7 User Interface

In the game there are a large number of windows. There is the character window which shows information about the attributes of the character, then the equipment window which shows the worn equipment and weapon. Moreover another window is the status window which shows the players health, food, gold and experience. Additionally there is the inventory window which shows the carried items, the player has with him. Furthermore another window is the sound and music button window and the in-game menu window and at last the shop window. All those windows will be shown and how they are done in the game.

```
function OnGUI (){
//should look the same at any resolution with the same aspect ratio
var rx : float = Screen.width / nativeWidth;
var ry : float = Screen.height / nativeHeight;
GUI.matrix = Matrix4x4.TRS (Vector3(0, 0, 0), Quaternion.identity,
Vector3 (rx, ry, 1));
GUI.skin = skin;
if(character_open == true){
    //update stats for displaying
    Gridcontent[0].text = "Strength    "+strength.ToString();
    Gridcontent[1].text = "Vitality    "+vitality.ToString();
    Gridcontent[2].text = "Planting    "+planting.ToString();
    Gridcontent[3].text = "Survival    "+survival.ToString();
    windowRect = GUI.Window (3, windowRect, DoMyWindow, "");
    //if Grid_button is clicked put points on it
    if((selGridInt == 0)&&(level_up_points>0)){
        strength +=1;
        level_up_points -=1;
        player.SendMessage("update_attack");
        strength_updated = true;
        selGridInt = -1;
    }
    else if ((selGridInt == 1)&&(level_up_points>0)){
        vitality +=1;
        level_up_points -=1;
        vitality_updated = true;
        player.SendMessage ("update_vitality");
        selGridInt = -1;
    }
    else if ((selGridInt == 2)&&(level_up_points>0)){
        planting +=1;
        level_up_points -=1;
        planting_updated = true;
        selGridInt = -1;
    }
    else if ((selGridInt == 3)&&(level_up_points>0)){
        survival +=1;
        level_up_points -=1;
        survival_updated = true;
        player.SendMessage("update_survival");
        selGridInt = -1;
    }
selGridInt = -1;
```

Figure 64. The `OnGUI` function of the “Character” script.

Figure 64 shows the `OnGUI` function of the “Character” script. The `OnGUI` function is called every frame of the game, so it is quiet similar to the `Update` function. Just the `OnGUI` function is for drawing the interface or other GUI elements. In the beginning of the function it is made sure, that all the UI elements adjust to the given resolution, which ensures they stay the same size in relation to the resolution. Then the skin is defined for this GUI. Additionally the `Update` function of this script checks if the “c” key is pressed for opening or closing the character window. If the window is open, it displays `Gridcontent` which is the content of a button array. So the content of each button of the array is defined with the String and the value which is transformed into a

string. The next line defines the window with a GUI window and the `DoMyWindow` function. Moreover the next four conditions are about the different attribute buttons. Each condition checks if its button has been clicked and then if there are levelup points, it adds a point to the attribute and calls the function with `SendMessage` to update the attributes as well in the “Mainchar” script.

```
164 function DoMyWindow (windowID : int)
165 {
166         GUILayout.Space(8);
167         GUILayout.Label("", "Divider");
168         GUILayout.Label("Character Level  " +level);
169         GUILayout.Label("", "Divider");
170         //buttongrid
171         selGridInt = GUILayout.SelectionGrid(selGridInt, Gridcontent, 1);
172         GUILayout.Label("Points left:  "+ level_up_points);
173         if (GUI.tooltip != "") {
174             GUI.Label(Rect(0, -50, w, 200), GUI.tooltip);
175     }
176 }
```

Figure 65. The `DoMyWindow` function of the “Character” script.

Figure 65 shows the `DoMyWindow` function in the “Character” script, which defines the elements of the window. At first it inserts space into the layout then it creates a label with the content “Divider”, which is defined in the skin and which is a line. Then another label is created with the “Character Level” and the value, which is closed then with another divider. Moreover the next line creates the button array and defines `selGridInt` which is used to define which button is pressed. Additionally there is another label with the remaining level up points. The last condition shows tooltips if hovered over the different buttons.

```
79      if(Equipment_open == true){
80          windowRect = GUI.Window (2, windowRect, DoMyWindow, "");
81          //if Grid_button is clicked unequips item
82          for(var i=0; i<size_equipment;i++){
83              if((selGridInt==i)&&(Equiped_items[i]!=null)){
84                  //if the inventory is not full
85                  if(player_inventory.is_full()== false){
86                      //if the equipement is a weapon put it out from the weapon slot.
87                      if(Equiped_items[i].GetComponent(Item).isWeapon == true){
88                          RemoveWeapon();
89                      }
90                      //place item back to the inventory
91                      Equiped_items[i].GetComponent(Item).Pickup_item();
92                      //unequip Item from character
93                      Equipment_off (Equiped_items[i]);
94                  }
95              }
96          }
97          selGridInt = -1;
98      }
99  }
100 //GUI window
101 function DoMyWindow (windowID : int)
102 {
103     GUILayout.Space(8);
104     GUILayout.Label("", "Divider");
105     GUILayout.Label("Equipment: ");
106     GUILayout.Label("", "Divider");
107 //buttongrid
108     selGridInt = GUILayout.SelectionGrid(selGridInt, Gridcontent, 3);
109 //show players defense
110     GUILayout.Label("Attack:  "+ player.GetComponent(Mainchar).damage);
111     GUILayout.Label("Defense:  "+ player.GetComponent(Mainchar).defense);
112     GUILayout.Label("Speed:  "+ player.GetComponent(Mainchar).speed);
113     if (GUI.tooltip != "") {
114         GUI.Label(Rect(0, -50, w, 200), GUI.tooltip);
115     }
116 }
```

Figure 66. Part of the `OnGUI` and `DoMyWindow` function of the “Equipment” script.

Figure 66 shows in the beginning the `OnGUI` function of the “Equipment” script. At first it checks if the window is open and then creates the window. Additionally it checks if one of the equipment buttons has been clicked and if the button has an equipped item. Next the code checks if the inventory is not full yet. If this is the case it checks if the equipped item is a weapon, then it calls the `RemoveWeapon` function, which removes the weapon. Additionally the pickup function of the item is called, which puts the item in the inventory and then the equipment is put off with the call of the `Equipment_off` function. The `DoMyWindow` function shows the configuration of the window with the headline label between two divider labels. Then there is again the button array and three additional labels with attack, defense and speed with the values from the “Mainchar” script. The tooltips are displayed when the mouse is hovered over the buttons.

```
69 if(stat_window_open == true){
70     hp.text = "Health   "+player.GetComponent(Mainchar).max_health.ToString()+" | "
71     +player.GetComponent(Mainchar).health.ToString() ;
72     food.text = "Food   "+player.GetComponent(Mainchar).max_food.ToString()+" | "
73     +player.GetComponent(Mainchar).food.ToString();
74     gold.text = "Gold   "+player.GetComponent(Mainchar).gold.ToString();
75     exp.text = "Exp   "+player.GetComponent(Mainchar).exp.ToString();
76     needed_exp.text = "Next   "+player.GetComponent(Character).exp_level.ToString();
77     windowRect = GUI.Window (0, windowRect, DoMyWindow, "");
78 }
79     windowRect2 = GUI.Window (20, windowRect2, button_window, "");
80 }
81 function DoMyWindow (windowID : int)
82 {
83         GUILayout.Space(0);
84         GUILayout.Label("", "Divider");
85         GUILayout.Label("Stats: ");
86         GUILayout.Label("", "Divider");
87         GUILayout.Label(hp.text);
88         GUILayout.Label(food.text);
89         GUILayout.Label(gold.text);
90         GUILayout.Label(exp.text);
91         GUILayout.Label(needed_exp.text);
92 }
```

Figure 67. Part of the `OnGUI` function of the “UI” script.

Figure 67 shows part of the `OnGUI` function which is for the status window in the game. If the window is open, the text strings are filled with the health, maximum health, food, maximum food, gold, current experience and the experience needed for next level. Then the window for the status window is created with the content of the `DoMyWindow` function. In this function all the strings which have been created before are written in the labels and displayed.

```
94 function button_window (windowID : int)
95 {
96 GUILayout.Label("");
97 GUILayout.Label("", "Divider");
98 //toggle sound
99 if(GUILayout.Button("Sound")){
100     if(sound_on == true){
101         //find audiosources
102         update_audio_sources();
103         //disable sound
104         toggle_all_sounds();
105         sound_on = false;
106         }
107     else{
108         //find audiosources
109         update_audio_sources();
110         //enable sound
111         toggle_all_sounds();
112         sound_on = true;
113         }
114     }
115 GUILayout.Label("", "Divider");
116 if(GUILayout.Button("Music")){
117     if(music_on == true){
118         //disable music
119         GetComponent(AudioSource).mute = true;
120         music_on = false;
121         }
122     else{
123         //enable music
124         GetComponent(AudioSource).mute = false;
125         music_on = true;
126         }
127     }
128 }
```

Figure 68. The `button_window` function of the “UI” script.

Figure 68 shows the `button_window` function in the “UI” script, which contains the sound button and music button window. In the beginning there are two empty labels and then there is a condition with a button with the label “Sound”. If this button is clicked the condition is fulfilled and toggles the sound. For toggling it is necessary to check if the sound is on or off at the time the button is clicked. Additionally the `update_audiosources` function is called, which updates all the audio sources and then the `toggle_all_sounds` function is called. Then another label divides the two buttons from each other. Moreover if the music button is clicked, it checks again if the music is on already and then gets the “Audio Source” of this game object and mutes or activates it.

```
130 function update_audio_sources () {
131 var position : int =0;
132 all_audio_sources = FindObjectsOfType(AudioSource) as AudioSource[];
133 var audio_sources = new Array(all_audio_sources);
134 //finds the audio source for music and removes it from the list
135 for(var i=0; i< all_audio_sources.Length; i++) {
136     if(all_audio_sources[i] == GetComponent(AudioSource)){
137         position = i;
138     }
139 }
140 audio_sources.RemoveAt(position);
141 all_audio_sources = audio_sources.ToBuiltin(AudioSource);
142 }
143 function toggle_all_sounds() {
144     if(sound_on == true){
145         for(var audio : AudioSource in all_audio_sources) {
146         audio.mute = true;
147         }
148     }
149     else{
150         for(var audio : AudioSource in all_audio_sources) {
151         audio.mute = false;
152         }
153     }
154 }
```

Figure 69. The `update_audio_sources` and `toggle_all_sounds` function.

Figure 69 shows the two functions, which have been used before. In the `update_audio_sources` function the code searches for all the “Audio Sources” in the game and saves them in an “Audio Source” array, which is then translated into the builtin array system. Additionally it searches for the “Audio Source” of this game object and puts it out of the array. This has to be done so that the music would not be disabled when the sound is disabled. Then in the next function it first checks if the sound is on and then it mutes all “Audio Sources” which are in the array. In case the sound is off, it will activate all “Audio Sources”.

```
71 for(var j=0; j<Gridcontent.length; j++){
72     if(player_inventory.Items[j].GetComponent(Item).stack > 1){
73         Gridcontent[j].text= player_inventory.Items[j].name +" "+ player_inventory.Items[j].
74         GetComponent(Item).stack.ToString();
75         }
76     else {
77         if(player_inventory.Items[j].GetComponent(Item).stackable == true){
78             Gridcontent[j].text = player_inventory.Items[j].name;
79             }
80         else{
81             Gridcontent[j].text = player_inventory.Items[j].GetComponent(Item).item_content.text;
82             }
83         }
84     }
85 if(inventory_open == true){
86     windowRect = GUI.Window (1, windowRect, DoMyWindow, "");
87 
88     for(var i=0; i<player_inventory.Items.length;i++){
89         if(selGridInt==i){
90             //check if it is a usable item
91             player_inventory.Items[i].GetComponent(Item_Effect).UseEffect();
92             }
93         }
94     selGridInt = -1;
95     }
96 }
```

Figure 70. `OnGUI` function of the “inventory_display” script.

Figure 70 shows the `OnGUI` function of the “Inventory_display” script. At first it checks if the items in the inventory array have stacks. If this is the case, then into the `Gridcontent.text` is written the number of the same items as a string. Then if the item is stackable but is only one item, it writes the name on the string, that there is no number displayed. Additionally when the display window is open, the window is created. Moreover in the next few lines it checks if any of the items is clicked and if it can be used and then calls the `UseEffect` function in the “Item_Effect” script, which was explained in section 4.3.2.

```
105 function add_content_grid (insert : GUIContent) {
106 var new_Gridcontent = new Array(Gridcontent);
107 new_Gridcontent.Add(insert);
108 Gridcontent=new_Gridcontent.ToBuiltin(GUIContent);
109 }
110 function remove_content_grid (content : GUIContent) {
111 var index = 0;
112 var new_Gridcontent = new Array(Gridcontent);
113 for(var i:GUIContent in new_Gridcontent){
114     if(i == content){
115         new_Gridcontent.RemoveAt(index);
116         break;
117         }
118     index++;
119     }
120 Gridcontent = new_Gridcontent.ToBuiltin(GUIContent);
121 }
122 function DoMyWindow (windowID : int)
123 {
124 GUILayout.Space(8);
125 GUILayout.Label("", "Divider");
126 GUILayout.Label("Inventory: ");
127 GUILayout.Label("", "Divider");
128 //buttongrid
129 selGridInt = GUILayout.SelectionGrid(selGridInt, Gridcontent, 5);
130     //displays the sold item + price
131     if(Item_sold == true){
132         GUI.Label (Rect (0, -50, 800, 200), "Sold Item "+item_sold_content.text+
133         " for "+ item_sold_value);
134         StartCoroutine(item_sold());
135     }
136         if (GUI.tooltip != ""){
137         GUI.Label(Rect(0, -50, 800, 200), GUI.tooltip);
138     }
139 }
```

Figure 71. Several important functions of the “inventory_display” script.

Figure 71 shows the code of other important functions of the “Inventory_display” script. The `add_content_grid` function adds a new item to the button array. This is called in the `Pickup_item` function, which has been explained in section 4.3.2. Then the `remove_content_grid` function removes one item from the button array. Moreover it is searched for the given content and if a match is found, the match is removed at the index and the array is rebuilt in the end. Then the last function `DoMyWindow` contains the layout for the inventory window. It has a headline with the name “Inventory” which is embedded between two “Dividers”. Furthermore the button array is drawn after that and if an item is sold, a text is written at the tooltip window, which is located at the top of the the inventory window.

```
53 function DoMyWindow (windowID : int)
54 {
55 GUILayout.Space(0);
56 GUILayout.Label("", "Divider");
57 GUILayout.Label("Options: ");
58 GUILayout.Label("", "Divider");
59
60 // Buttons
61     if( GUILayout.Button("Back to Main Menu")== true){
62         Time.timeScale = 1.0;
63         Application.LoadLevel("Mainmenu");
64     }
65     if(GUILayout.Button("End Game")){
66         Application.Quit();
67     }
68     if( GUILayout.Button("Cancel")== true){
69         options_window_open = false;
70         Time.timeScale = 1.0;
71     }
72         GUI.DragWindow (Rect (0,0,10000,10000));
73 }
```

Figure 72. The `DoMyWindow` of the “Optionsingame” script.

Figure 72 shows the `DoMyWindow` function of the “Optionsingame” script. This script creates the in-game menu, which pauses the game when it is opened. This figure shows the main functionality of the script. As before it has “Options” as headline embedded of two “Dividers”. Then there are three buttons. The first is the main menu button which stops the break and sets the time back to one and uses the function `Application.LoadLevel` which calls the scene with the name defined in brackets. Additionally the second button is the end game button. When the button is clicked, it just calls the `Application.Quit` function, which is an Unity integrated function for exiting the game. The last button is the cancel button, which closes the window and continues the game. Moreover the last line of code makes the window draggable.

```
59      windowRect = GUI.Window (6, windowRect, DoMyWindow, "");
60      //access each button from the grid
61      for(var i=0; i< selGridcontent.length; i++ ){
62          if ((selGridInt == i)&&(player.GetComponent(Mainchar).gold >= items[i].GetComponent(Item).
63              buy_value)&&(player_inventory.is_full()== false)){
64              var clone : GameObject = Instantiate(items[i]);
65              clone.name = items[i].name;
66              clone.GetComponent(Item).Pickup_item();
67              player.GetComponent(Mainchar).gold -= items[i].GetComponent(Item).buy_value;
68              //plays sound when item is bought
69              play_audio ();
70              StartCoroutine(change_tooltip_back(selGridInt, " You bought " + items[i].name + "for "
71                              + items[i].GetComponent(Item).buy_value));
72              selGridInt = -1;
73          }
74          else if ((selGridInt == i)&&(player.GetComponent(Mainchar).gold < items[i].GetComponent(Item).
75          buy_value)) {
76          selGridInt = -1;
77              }
78          }
79      }
80  }
```

Figure 73. Part of the `OnGUI` function of the “Shop” script.

Figure 73 shows the important part of the `OnGUI` function of the “Shop” script. At first the code creates again the window and then it checks which button of the button array is clicked by the player, which means which item the player wants to buy. If the player has enough money and the item of the player is not yet full, it creates a clone of the item which has been clicked. Additionally the `Pickup_item` function is called, which adds this item to the inventory array. Next the player’s gold is reduced by the price of the item and a sound is played. Moreover the tooltip is changed and it shows for a short time the item which has been bought and how much gold it cost. In the end the button is again deactivated as well if the player did not have enough money. Furthermore in this script it is checked if the player is in the shop or not and if he presses the “b” key, the shop will open.

This section gave an overview over the UI and how it was implemented in this game. In the next section the different game scenes briefly and how they were used will be explained.

4.8 Game Scenes

The game scenes are an essential part of the game. In this game there are six scenes which are used. There is the game scene, which is the main part of the game. All the game related content is in this scene. Then there is the Instructions scene, which explains the game mechanics to the player with text and pictures. Additionally there is the Intro which is another scene and displays the introduction story after starting the game. Then there is the win and lost screen which have also their own scenes, which are changed to when the game is over. Moreover the last scene is the main menu scene, which is the start screen of the game with three buttons start game, instructions and quit game.

5 Results and Discussion

The result of this project is a working Unity 2D game, which was developed for the Windows platform. The project itself took approximately six months, included with finding software, platform, graphics, sound, music and tools. Moreover doing tutorials and learning how to use the Unity engine and third party tools such as 2D toolkit and Navmesh2D were included in this time. In this chapter some pictures of the game will be shown and advantages and drawbacks of using Unity 2D for development of a 2D game.

For a better insight in the development process of this game project figure 74 shows the whole workflow in a Gantt chart.

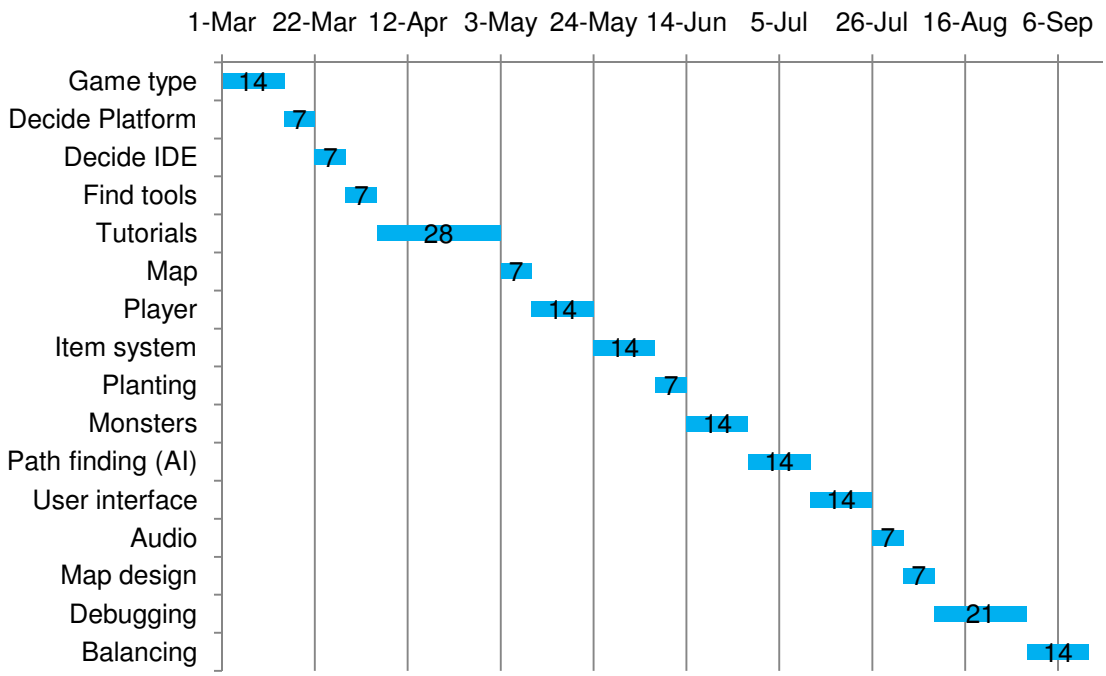

Figure 74. Gantt chart of the project.

Figure 74 shows the Gannt chart of the development progress of the game project. It began in March with ideas of the game and the platform. Moreover it took quite an amount of time for me to actually start with the project itself. After approximately two months of planning and doing tutorials the real project started with first creating the map. That was followed mainly by the programming for the player and the inventory system and other related features. Next was the Planting system and Monster implementation which took quite some time. Furthermore path finding took a longer time than expected even with the use of a third party tool. User interface and audio were slightly difficult to implement. In the end after the map design was done debugging and balancing took a longer period, even I was always debugging during the whole development process. It took approximately half a year to finish the project. Next the game itself will be shown with the main menu and a short insight in the ready game.

Figure 75. Main menu of the game.

Figure 75 shows the main menu of the game Harvest Survive. It has three buttons. With Start Game the game will switch to the Intro and then start the game. If the instructions button is clicked it will change to the instructions of the game and explain the player the basics of the game in text and pictures. Moreover if the end game button is clicked, the game will exit and return to windows.

Figure 76. The final version of the game.

Figure 76 shows the final version of the game. In the center there is the player with a green health bar floating over his head. Then there are mountains around which block the player's and monsters' movement. The bat in the map is a monster which is hunting

the player and has a red health bar over its head. At the bottom there is the GUI, which contains the character window, equipment window, status window, inventory window and sound window. Each of them has been explained in section 4.7. The player starts with only a stick and two carrot seeds and 100 gold coins. With this he has to grow the plants and sell the harvested fruit to the shop to earn money. With enough money it is possible to buy weapons and armor which will help to fight the monsters and bosses. The game has six different zones and for entering each zone a boss has to be defeated. The final boss is waiting in the last zone and if he is defeated, the player will win the game.

The background story is as follows. A farmer lives alone with his sister in a mysterious kingdom. There is a legend going around that if a person is at the brink of death, death himself will come to visit and give a choice to die soon, or try to challenge him in his own world. The farmer never believed it to be true, but one day when his sister was out selling her goods a dark traveller knocked his door and told him he is death and he asked him what he would choose. He decides to stay alive for his sister's sake, who would have a hard time living alone as a farmer. So you accept the challenge and enter the realm of death. Death created the world so, that the farmer's skills which is a vast knowledge of farming can be used. Death said he would wait for the farmer in the last area of the world and if he managed to defeat him he would live a long life. With this promise in mind the farmer starts to walk around in this unknown world.

Of course in the development process there were various problems which were solved and a large number of bugs which were fixed. In the following the biggest problems, which occurred in the development of the game will be discussed. In the beginning the first major issue was to create the map without overlapping sprite, in other words a tilemap. This feature is not included in Unity 2D, so it was difficult to create a more feature-rich map with it. That is the reason toolkit2D was used to solve this problem. This tool solved the problem with creating tilemaps and made it easy to edit them. However of course every third party tool had to be learned and especially if the documentation and examples lacked information. So it was difficult to access a single tile of the tilemap, which was quite a big issue.

Another major problem was the pathfinding for the monsters in the game, especially because there were a large number of mountains and other objects which blocked the monsters and player from moving over them. At first I tried to use the in-built navigation

system from Unity, but it is not working with Unity 2D, which made it necessary to use another third party tool. The tool was Navmesh2D, which used the built-in pathfinding system from Unity and is based on the A* algorithm. Additionally with using the Navmesh2D tool there was another issue regarding the programming, because I used JavaScript for most programming. However because the whole code of Navmesh2D was in C# and accessing this code from JavaScript and the other way around is quite difficult, I had to use also C# scripts just for the pathfinding. Moreover there was a problem with my pathfinding script, which made the monster follow the player but always calculated a new path when the player moved. So the player could easily move in circles around the player while the monster would be stuck calculating the new path. For solving this problem I made that the monster calculate the path after a certain delay, which made the monster move fluently after the player.

A further problem was making the weapon animations. There was only one weapon sprite for each weapon, so I had to move and turn them with a graphic tool but with rotating pixels the weapon pixels blurred. There was not a really good solution for this problem, so basically all sprites for the animation would need to be created from scratch or copy single or multiple pixels from the graphic and paste it to create the new sprites of the animation. Moreover with creating this big map and each area with monsters spawning and walking around led to a huge performance issue. As solution only in the area the player is at that moment the monsters spawn and move around, which made the game fluent.

The next problem which occurred was the inventory system I created. For this system I used the selectiongrid from built-in Unity, which is basically an array of buttons. So for each item added there was another button. The only issue with this approach was that it only supported a left click but no right click or other mouse buttons. This limited some functionalities and made them more complicated. Finally, an issue also was the collision detection for the fighting system. Basically the collision happens when the weapon touches the monster, but this happens of course a large number of times each second, because the collision is called by every frame. So it was important to limit the detection to a certain amount of time, which was quite difficult. In the end I used several coroutine functions to solve this problem.

As regards Unity 2D and its advantages and disadvantages for developing a 2D game, Unity 2D is going the right way with the first support for native 2D games. It comes al-

ready with many tools, such as the easy sprite editor, the 2D physics and collision detection, the 2D animation tool Mecanim and the 2D camera. All those tools are very useful in developing a 2D game with Unity. However there are still some features missing which should be included in a 2D game engine. The missing features are especially the tilemap feature and the navigation system, which are not yet supported by Unity 2D. Those two features are very essential for developers, since they save time in creating games. Moreover if these features are not included it can easily become expensive to pay for third-party tools, which provide the lacking features. So in the current state, if there is a need for pathfinding or tilemaps and the game should be programmed without any costs for the programming part, it will probably be better to look for an alternative. Otherwise Unity is quite a reliable game engine, which will probably be also a good choice to develop 2D games with in the future.

6 Conclusion

In this thesis I showed the game mechanics of the 2D Unity game I developed and I gave insight into programming with Unity 2D and the advantages and disadvantages to program with this game engine. Furthermore the result of the project was a working Unity 2D game for Windows. Moreover developing this game project in half a year's time, gave a good insight into the game development process. Especially it showed me how important it is to plan well before starting the project and choosing the platform and the game engine wisely. No game engine is the best and it should be chosen according to the developer's needs. That means it is important to collect much information about the engines before choosing one of them and get already some idea if the game project is possible with these tools or not and if there is a need for additional third-party tools. The more planning is done before starting the project, the less chance of failure for the project and less problems will arise during the development process.

As regards Unity 2D, I am sure it will be further developed and improved, so perhaps at some point there will be less need for third party tools. This would make the development easier and more efficient.

References

1 Unity Technologies. Unity [online]. San Francisco United States, Unity Technologies, 2 August 2014.
URL: http://unity3d.com/unity.
Accessed 2.8.2014.

2 Corazza, Seraphina. History of the Unity Engine Freerunner 3D Animation project [online]. Seraphina Corazza, 14 February 2013.
URL: http://seraphinacorazza.wordpress.com/2013/02/14/history-of-the-unity-engine-freerunner-3d-animation-project/
Accessed 2.8.2014.

3 Unity Technologies. Integrated Editor [online]. San Francisco United States, Unity Technologies, 2 August 2014.
URL: http://unity3d.com/unity/workflow/integrated-editor.
Accessed 2.8.2014.

4 Unity Technologies. Asset Workflow [online]. San Francisco United States, Unity Technologies, 2 August 2014.
URL: http://unity3d.com/unity/workflow/asset-workflow.
Accessed 2.8.2014.

5 Unity Technologies. Scene Building [online]. San Francisco United States, Unity Technologies, 2 August 2014.
URL: http://unity3d.com/unity/workflow/scene-building.
Accessed 2.8.2014.

6 Unity Technologies. Rapid Iteration [online]. San Francisco United States, Unity Technologies, 2 August 2014.
URL: http://unity3d.com/unity/workflow/rapid-iteration.
Accessed 2.8.2014.

7 Unity Technologies. Scripting [online]. San Francisco United States, Unity Technologies, 2 August 2014.
URL: http://unity3d.com/unity/workflow/scripting.
Accessed 2.8.2014.

8 Unity Technologies. Networking [online]. San Francisco United States, Unity Technologies, 2 August 2014.
URL: http://unity3d.com/unity/workflow/networking.
Accessed 2.8.2014.

9 Unity Technologies. Rendering [online]. San Francisco United States, Unity Technologies, 2 August 2014.
URL: http://unity3d.com/unity/quality/rendering.
Accessed 2.8.2014.

10 Unity Technologies. Lighting [online]. San Francisco United States, Unity Technologies, 2 August 2014.
URL: http://unity3d.com/unity/quality/lighting.
Accessed 2.8.2014.

11 Unity Technologies. Special Effects [online]. San Francisco United States, Unity Technologies, 2 August 2014.
URL: http://unity3d.com/unity/quality/special-effects.
Accessed 2.8.2014.

12 Unity Technologies. Terrains [online]. San Francisco United States, Unity Technologies, 2 August 2014.
URL: http://unity3d.com/unity/quality/terrains.
Accessed 2.8.2014.

13 Unity Technologies. Audio [online]. San Francisco United States, Unity Technologies, 2 August 2014.
URL: http://unity3d.com/unity/quality/audio.
Accessed 2.8.2014.

14 Unity Technologies. Physics [online]. San Francisco United States, Unity Technologies, 2 August 2014.
URL: http://unity3d.com/unity/quality/physics.
Accessed 2.8.2014.

15 Unity Technologies. AI [online]. San Francisco United States, Unity Technologies, 2 August 2014.
URL: http://unity3d.com/unity/quality/ai.
Accessed 2.8.2014.

16 Unity Technologies. Animation [online]. San Francisco United States, Unity Technologies, 2 August 2014.
URL: http://unity3d.com/unity/animation.
Accessed 2.8.2014.

17 Unity Technologies. 2D-3D [online]. San Francisco United States, Unity Technologies, 2 August 2014.
URL: http://unity3d.com/unity/2d-3d.
Accessed 2.8.2014.

18 Unicron Software. 2dtoolkit [online]. Newcastle England, Unicron Software, 2 August 2014.
URL: http://www.unikronsoftware.com/2dtoolkit/
Accessed 2.8.2014.

19 Pigeon Coop. Navmesh2D for Unity [online]. Melbourne Australia, Pigeon Coop, 5 March 2014.
URL: http://forum.unity3d.com/threads/navmesh2d-navmesh-generation-and-navigation-for-your-2d-projects-released.231022/
Accessed 2.8.2014.

20 Unicron Software. Tilemap Tutorial [online]. Newcastle England, Unicron Software, 2 August 2014.
URL: http://www.unikronsoftware.com/2dtoolkit/docs/latest/tilemap/tutorial.html
Accessed 2.8.2014.