FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Peter Guba

Cube Fighter

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: RNDr. Jan Kofroň, Ph.D.

Study programme: Computer Science

Specialization: Programming and Software systems

Prague 2019

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In…..... date..... signature of the author:

I would like to thank my supervisor, RNDr. Jan Kofroň, Ph.D, for his time, advice and his always pleasant approach.

Title: Cube Fighter

Author: Peter Guba

Department / Institute: Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: RNDr. Jan Kofroň, Ph.D., Department of Distributed and Dependable Systems

Abstract: Real-time strategy games and action games are two very different types of computer games. In my project, I attempted to see whether they could be combined to make a coherent whole, with mixed results.

The backbone of my game is its real-time strategy mode, inspired by Warcraft III and The Battle for Middle-earth 1 and 2. In this mode, the player can create buildings and units and use them to fight the enemy.

The game also has another mode of playing which imitates action games. In it, the player only controls a single unit.

The second mode ended up being just a fun add-on instead of an essential part of the gameplay as I had originally intended. The reason behind that wasn't that it would be impossible to make it so, but rather that it would require my game to be far more complex.

Keywords: real-time, strategy, RTS, action, game

Název práce: Cube Fighter

Autor: Peter Guba

Katedra / Ústav: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: RNDr. Jan Kofroň, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Realtimeové strategické hry a akční hry jsou dva velice rozdílné typy počítačových her. V mém projektu jsem se pokusil zjistit, jestli je možné je zkombinovat do jedné, se smíšenými výsledky.

Základem mé hry je její realtimeový strategický mód, inspirovaný Warcraftem III a The Battle for Middle-earth 1 a 2. V tomhle módu dokáže hráč tvořit budovy a jednotky a použít je na boj s nepřítelem.

Hra má také druhý mód, který imituje akční hry. V tomto módu hráč ovládá pouze jednu jednotku.

Druhý mód se nakonec stal jenom zábavným dodatkem, a ne esenciální součástí hry, jak jsem si původně představoval. Důvodem nebyla nemožnost realizace, ale její komplexnost.

Klíčová slova: realtime, strategie, RTS, akční, hra

Contents

1. Introduction **2**

2. Analysis **3**
2.1. Language conventions 3
2.2. RTS games 3
2.3. Action games 4
2.4. My implementation 4

3. Game design **8**
3.1. The Unity engine 8
3.2. Basic GameObjects of the game 11
3.3. Buildings and associated objects 13
3.4. Units 15
3.5. Icons 17
3.6. Weapons 19
3.7. Other GameObjects 19

4. Developer documentation **21**
4.1. Unity functions 21
4.2. GameMaster 25
4.3. Basic scripts 26
4.4. Buildings 29
4.5. Units 33
4.6. A.I. 37
4.7. Player 40
4.8. Icons 43
4.9. Weapons 45
4.10. Spells 48
4.11. Other scripts 49

5. Conclusion **51**
5.1. Fulfilment of goals 51
5.2. Possible Improvements 52

Attachments **54**
Attachment A: References 54
Attachment B: Archive description 55
Attachment C: User documentation 56

1. Introduction

As for many other people my age, computer games are an inseparable part of my childhood. I used to play all kinds of games, from Donkey Kong to The Battle for Middle-earth. Because of this, I decided to try and create a game which combines aspects of two types of computer games that are my favourites – strategic games and action games.

However, my aim in this thesis was not to create a big game with a broad range of spells, items, and characters that could compete with other games. That would have been borderline impossible, given that these games are usually developed by teams of experts over years. Instead, my goal was to create an easily expandable skeleton of such a game.

More specifically, the goals of my project were:

1.) Create a basic real-time strategy game with a few types of units and buildings.
2.) The game should include one human player and at least one A.I.
3.) Provide the player with the option of playing for any fighter unit individually, as if he/she were playing an action game.
4.) When commanding units individually, the combat system should be interesting enough to be entertaining.
5.) Implement a system of casting spells.
6.) Implement a system of collecting and using items.
7.) Make the whole game easily expandable. Adding new items, spells, units, etc. should be straightforward.

2. Analysis

In this section, I first go over the structure of real-time strategy games (RTS games for short) and action games, and then I describe the structure of my own game. However, seeing as both of these genres are very broad and probably contain hundreds or even thousands of different games, it is impossible to discuss them in detail. I therefore only go over the most basic concepts that are either almost universally present in the genre, or important for the purposes of this thesis.

2.1. Language conventions

Before getting into the analysis, I have to explain some basic terms and concepts to avoid misunderstandings. First, the word ‘player’ could be used to refer to both the human playing this game and any agent acting in the game (be it a human or an A.I.). To avoid confusion, I refer to the latter kind of player as an agent.

Next, when it comes to creative things like games, it is almost impossible to say that something holds for every example of a given type of game, musical style, etc. Therefore, any time I say that something is done a certain way in RTS or action games, I mean that it is so in most cases.

The last thing I need to mention is that one of the aims of my project is easy expandability, so even though the possibilities of the actual game are quite limited, I often use more general terms.

2.2. RTS games

2.2.1. Design & controls

In RTS games, the screen is divided into a map area displaying the game world and a minimap which provides an overview of the entire map. An agent is provided with a view of the world which, in more modern games, is done via a free-roaming camera which provides an aerial point of view. Players mainly scroll the screen and issue commands with the mouse or using keyboard shortcuts.

2.2.2. Entities

There are two basic types of entities in RTS games – units and buildings. There are usually one or more types of units which can be used for creating buildings and/or gathering resources, while the other types are used in combat, either as direct fighters or as support units.

Some RTS games have special types of units, often called ‘heroes’, of which there can always be only one unit alive at a time and which are more powerful than other units. These, and sometimes also other units, can level up by fighting.

Buildings are mostly used for generating different things like units, upgrades, or resources, although some can be used for protection or offense.

2.2.3. Gameplay

The gameplay consists of the player (and other agents, human or artificial) being positioned somewhere on the map with a few units or a building. These allow the agent to create more units and buildings. Often, but not always, an agent must build specific structures to unlock more advanced units and structures. RTS games require an agent to build an army, the normal size of which can vary from less than 10 to hundreds of units. This army is used to either defend the agent from attacks or to eliminate enemies who possess bases with unit production capabilities of their own.

Resource gathering is often the main focus of RTS games, but some titles of the genre place higher gameplay significance on how units are used in combat. Some games impose a ceiling on the number of simultaneously existing troops, while other titles have no such unit cap.

2.3. Action games

In action games, players control a single character and use it to advance through levels that present them with challenges with increasing difficulty. The player's avatar has various abilities and means of overcoming these challenges.

Often, the player can collect different items which can either be used for some specific purpose (like unlocking a door), or which temporarily or permanently improve some of the avatar's abilities.

The avatar has some sort of hit-marker or health which can be depleted by enemies or obstacles and, once it runs out, the avatar dies. The avatar usually has some number of lives, which allow the player to start over from a checkpoint or the start of a level.

In 3D action games, the player's perspective is tied to the avatar from either first- or third-person perspective. In these games, the player can control the avatar by changing its direction with the mouse and using the keyboard to move it. The mouse is also often used to attack or perform some other action by clicking one of its buttons or scrolling its wheel.

Action games tend to keep track of the player's score, which can be gained by completing challenges or killing enemies.

2.4. My implementation

2.4.1. Application design

When designing this game, my primary aim was to create a playable RTS game. My secondary aim was to add elements of action games to it. This resulted in this game having two modes of playing – strategic mode and 3rd person mode. In the strategic mode, the game is like most other RTS games. The player is provided with an aerial

view and a minimap and can command units with the mouse. There is also a menu in the lower right corner of the screen which can display various clickable icons. For example, the buildings that a worker can build are displayed there.

In 3rd person mode, the camera through which the player sees is tied to a single unit and the player loses control of all other units. On the other hand, the player gains access to the unit's spells. Also, the unit's attacks stop being automated. Instead, the player has to trigger each attack him-/herself. Playing in this mode is similar to playing an action game, however, contrary to typical action games, there is no protagonist unit in this game. Because of this, I omitted some of the game aspects that are often found in action games. Each unit has only one life for example.

2.4.2. Rules

In this aspect, the game is a watered-down version of the RTS games that inspired me (Warcraft III and The Battle for Middle-earth I and II). The player starts out with a single building, called the Base, which allows him/her to create units called Workers. Those in turn allow him/her to create other buildings which fulfil different functions. The goal is to destroy all the enemy's buildings and units.

I decided to forgo implementing any methods of resource gathering by units and instead opted for the approach used in The Battle for Middle-earth – having resource-generating buildings.

The 3rd person mode isn't necessary or even essential for winning the game, so it can be played just like a classic RTS. However, the more detailed coordination of a single unit that the mode provides can give the player an advantage.

2.4.3. Gameplay

The essence of every game is presenting the player with some sort of challenge and giving him/her some means of overcoming it. In my case, the challenge is to defeat the enemy army and destroy their buildings. The means of overcoming the challenge is making your own army and buildings. This forms the backbone of the gameplay.

Besides the mentioned factors, there could be a lot of other aspects of the game that could influence the gameplay. For example, there could be different kinds of environments and different kinds of resources to be gathered. I didn't include these in my game because I didn't see them as essential to the kind of game I wanted to create. Something I did implement though (or the basis of it at least), is the gathering of items and the concept of spells.

The actual number of items in my game is very limited – there are three weapons and two potions. However, the functionality of gathering and using/equipping items is fully implemented and, while not being an essential part of the gameplay, gathering the potions can help one win.

As far as spells go, every unit has two, although the spellbar has space for ten. Once again, I didn't implement more, because my primary goal was implementing the functionality of adding and using spells.

I also implemented the concept of gathering experience. Every fighter unit can gather experience by destroying enemies. When it gathers a certain amount, it levels up, which increases its power, amount of health, and other properties. Once it levels up, the amount of experience necessary to level up again increases by 50.

2.4.4. Programming language & environment

Games can be programmed in any programming language, although some are better suited for it than others. The best would probably be C++, seeing as it is quite a low-level language. That allows for quicker execution of code which is essential for games. However, my foremost criterion when choosing the programming language that I would create this project in was, that I wanted to have some experience with it to avoid unexpected setbacks caused by a lack of understanding of the language. This limited my possibilities to just three languages: C#, Java and Pascal. Pascal lags behind the first two in just about every aspect though, so I was really just choosing from the first two.

Between those two choices, it came down to what I'd had more experience with at first. That is because the languages are quite similar as far as both semantics and performance go, although C# has a bit of an edge on Java in the latter aspect. For this reason, I first decided on Java and looked for a game engine that would allow me to create a 3D game programmed in that language.

I found jMonkeyEngine [1], a free and open-source game engine made especially for modern 3D game development in Java. It is a collection of libraries, but an official SDK exists for it, which is based on the NetBeans platform. This platform seemed to have the necessary tools for creating this project – a physics engine, tools for creating a 3D environment and 3D entities to place in that environment, and, most importantly, tools for programming the behaviour of those entities. However, I found the documentation lacking, and since it isn't such a well know engine, there wasn't much information to be found about it elsewhere.

I then switched to the Unity engine [2], for a few reasons. First, Unity is one of the most used and well-known game development softwares in the world and it allows the user to create all manner of games, both 2D and 3D. That means that it both provides the means for creating a basic RTS, such as the one I was aiming for, and that there is an abundance of information about it on the Internet, both official and unofficial. Second, it is very user friendly. And third, my brother had already had some experience with it, so I had someone to show me the ropes.

Unlike jMonkeyEngine, the primary development language in Unity is C#, but that still fit with my previous requirement for a language which I'd had experience with. There are also other languages that can be used to make games in Unity, like UnityScript (derived from JavaScript) or Boo, but C# is definitely *the* language of Unity, since all of its libraries are programmed in it. This is because Unity uses Mono, a cross-platform implementation of Microsoft's .NET, of which C# is the primary language.

Besides Unity, there were many other well-known engines that I could have used. For example, the Unreal Engine [3], which you can find topping many lists of best game engines (with Unity often close behind). A great advantage of this engine is that it uses C++ which, as I've said before, is the language best suited for making video games. Other notable examples include GameMaker, Godot, CryEngine, and many more, however, all of these either aren't as user friendly as Unity, or don't use either of the two languages I was looking for.

2.4.5. Aesthetics

The aesthetics of my project are a result of the possibilities provided by Unity. When I started designing this game, I first tried downloading units, buildings, and other things from the Unity asset store [4]. However, manipulating models that I didn't design myself proved to be quite challenging, especially given the fact that I was a complete novice at Unity. Another issue was that the game's aesthetics would be inconsistent unless I found some package which contained all the objects that I needed.

In the end, I decided to create my own objects. These objects are just comprised of basic geometric shapes offered by Unity, like cubes and spheres. However, since they were created by me, it was easy for me to work with them. They are also aesthetically consistent and tailored to the specific requirements of my game. One downside of this approach though, is the fact that some objects aren't rendered quite properly. I believe this is a fault in the Unity environment, not in my design.

3. Game design

3.1. The Unity engine

Like any good game engine, Unity supplies its user with the tools to create games quickly and effectively. It provides an environment in which the user can create various 2D and 3D objects and assign them properties and behaviours. In this section, I go over the basic tools that Unity has to offer. More information about any of these components can be found in the Unity user manual [5].

3.1.1. GameObjects

According to the Unity documentation, "the GameObject is the most important concept in the Unity Editor". Every object in a project is a GameObject.

Every GameObject has two important pieces of information associated with it – a tag and a layer. Tags are arbitrary strings that the user can use to identify objects. Layers are sets of objects which should share some common characteristic, although this is not enforced in any way by the engine. They are used to restrict operations like raycasting only to relevant objects.

The user can combine different GameObjects into hierarchies, forming larger, more complex GameObjects.

GameObjects can fulfil a variety of different functions, but by themselves they are blank canvases. To add functionality to them, you need to add components.

3.1.2. Components

Components are the functional pieces of every GameObject, they define their properties and behaviours. An important property of some components – most notably the Script component – is that they can be disabled. This allows the properties/behaviours which they grant the GameObject to be turned on/off while the game is running.

In this section, I describe the most important of the many components that Unity has to offer.

Transform

The Transform component determines the position, rotation and scale of every object in the scene. Every 3D GameObject has this component.

Rect Transform

A 2D counterpart of the Transform component. While Transform represents a single point, Rect Transform represents a rectangle that a UI element can be placed inside.

Collider

This component defines the shape of an object for the purposes of physical collisions. A Collider is invisible and it isn't necessarily the same shape as the object to which it is attached. All the Colliders that I have used in this project are of the primitive kind – Box, Sphere, and Capsule Colliders – with the exception of the Terrain Collider.

Colliders can interact with each other in different ways, based on their settings as well as the settings of the Rigidbody component (discussed next), if one is attached. For the purposes of this thesis, it suffices to mention one important property that influences this behaviour – the isTrigger boolean flag. If a Collider has this flag set to true, it only detects when it intersects another Collider. If it is set to false, intersecting with another Collider creates a collision which can move the objects to which the two Colliders belong. The former kind is called a trigger Collider, while the latter is called a non-trigger Collider.

Rigidbody

A Rigidbody is a component that simulates physical behaviour for a GameObject. It goes hand in hand with Colliders – there are very few cases in which a GameObject should have a Collider but not a Rigidbody (one such exception is a static, immovable object, like a wall). Rigidbodies allow GameObjects to be moved by Collisions detected by the Colliders. If it is necessary for an object to detect Collisions without being moved by them, the isKinematic flag of the attached Rigidbody should be set to true.

Mesh Filter & Mesh Renderer

These two components allow the object to be rendered inside the scene. The Mesh Filter defines the geometry of the object, while the Mesh Renderer takes and renders it at the position defined by the Transform component.

Nav Mesh Agent

A Nav Mesh is a data structure which Unity generates to approximate the walkable areas of the environment. A Nav Mesh Agent uses this data structure to navigate the world and avoid obstacles. Nav Mesh Agents are also capable of avoiding each other.

Nav Mesh Obstacle

This component allows the user to describe moving obstacles that Nav Mesh Agents should avoid.

Material

As the name suggests, this component defines the visual properties of the object to which it is attached.

Script

A core component for any game. A script is a piece of code. It allows the user to define the behaviours of GameObjects. For a script to be attachable as a component however, it needs to inherit from a built-in class called *MonoBehaviour*. This allows the programmer to use many useful Unity functions.

Seeing as scripts allow the user to define the behaviours of GameObjects, I discuss them extensively in the following chapter. Therefore, I need some way of distinguishing components and GameObjects from scripts, in case they have the same name. To this end, all the names of scripts, classes, methods and variables are written in cursive.

Canvas

A Canvas is a component which defines the area that all UI components, like Buttons, Sliders, or text fields, should be inside. Through its Render Mode setting, it can be made to always overlay the currently used Camera, as if it was stuck onto it (Screen Space - Overlay), to always be placed in front of a given Camera (Screen Space - Camera), or to exist independently of Cameras (World Space).

When a Canvas GameObject (one of the predefined GameObjects in Unity) is created, it has more than just the Canvas component attached. It also has the Rect Transform, Canvas Scaler and Graphic Raycaster components. The function of the latter two is unimportant for the purposes of this thesis though. When I mention a Canvas GameObject, assume that all of the mentioned components are present.

Canvas Renderer

Renders a graphical UI object contained within a Canvas.

Camera

A Camera is exactly what one would expect – a component that captures and displays the world to a player. An important property of this component is the Culling Mask. This specifies the layers which the Camera is supposed to capture. If the culling mask of a Camera doesn't include a GameObject's layer, the Camera doesn't capture the GameObject.

When a Camera GameObject (another predefined Unity GameObject) is instantiated, it also has an Audio Listener component attached. This component is an imitation of a microphone. It records the sounds around it and plays them through the speakers. Only one Audio Listener can be active in a scene at any given time.

Light

Unity offers many kinds of lights – Point Light, Area Light, Directional Light, etc., the properties of which can differ. There are some properties that they all share though, and an important one of those is the Culling Mask, which describes which layers should be illuminated by this light.

Animator

This component allows the user to assign animations to a GameObject.

3.1.3. Assets

Besides its native 3D objects, Unity also supports using and, in some cases, even creating many additional assets. These include images, FBX models, animations, and audio files. Unity also comes equipped with a package of standard assets, like Cameras, effects, and Particle Systems, that are widely used by many users.

3.2. Basic GameObjects of the game

Before getting into the details of my implementation, I once again need to clear up some terminology. In the previous section, I explained some of the components that Unity has to offer. I refer to these often in the following pages. The problem is that, for example, the name ‘Camera’ can be used to describe both the component and the GameObject to which it is attached. In order to avoid confusion, I use the latter meaning. Any exception to this rule is explicitly stated.

3.2.1. DirectionalLight

A kind of Light which behaves as if the rays were coming from a source that is infinitely far away, so they travel in parallel which makes it a good approximation of the sun. It lights up the entire game world.

3.2.2. Terrain

The map on which the game is played is just one big GameObject called Terrain. It has a Terrain Collider but no Rigidbody, since it isn’t supposed to be able to move. It also has a special Terrain component which renders it.

3.2.3. EventSystem

Takes care of sending events to objects based on input. An object of this kind is generated automatically when the user adds a Canvas.

3.2.4. GM

GM is short for game master. This object controls the initialization, pausing and resetting of the game. When the application starts, the *GameMaster* script in the root element of the GameObject's composition hierarchy displays the Main menu. The functionality of all the buttons in this and every other menu is implemented in that script.

GM is comprised of several parts. First, there is the root element, which has just two components – a Transform component and the *GameMaster* script. Then there are the menus. They are all direct children of the root element and are in turn comprised of a Canvas, which is in the root element of the menu (not to be confused with the root element of GM), and buttons or other UI elements, all of which are children of the menu root element.

The rest of the GM GameObject is made up of four GameObjects that only have the Transform component attached. They are the starting points for the player, the A.I., and two groups of Monster units which don't belong to either team.

3.2.5. MapCamera

This is a stationary Camera positioned high above the Terrain. Its role is to capture the whole Terrain. What it sees is then projected onto a texture in the lower left corner of the screen, forming a minimap. Its Culling Mask is set to only include the 'Minimap' layer. This is a layer that all GameObjects that should only be visible on the minimap are supposed to have assigned.

3.2.6. StrategicCamera

This is the only player-controlled Camera, which isn't connected to a specific unit. It provides an aerial view of the world which allows the player to see enough of the world to be able to make strategic decisions.

This GameObject is comprised of two elements – the root element, to which the Camera component is attached, and a child element called MinimapLight. This element has a Light component attached to it, the type of which is Spot and the Culling Mask of which is set to the 'Minimap' layer. The result is that this light is only visible on the minimap, the utility of which is that the Player is able to see where on the map the Camera currently is.

3.2.7. MinimapTerrain

This is a copy of the Terrain GameObject with 2 differences. The first is that MinimapTerrain doesn't have a Terrain Collider component. The second is that the layer to which it is assigned is 'Minimap', as opposed to the 'Terrain' layer, to which Terrain is assigned. The reason behind that is the reason why I created this GameObject in the first place – it is a copy of the Terrain, which is illuminated by

the Light from the Camera currently being used, letting the player know on which part of the world he/she is looking.

3.2.8. Music

This is a GameObject to which an Audio Source component is attached. This is a component capable of producing sound. In this case, it is responsible for playing music.

3.3. Buildings and associated objects

All the buildings have these components attached to their root elements: Transform, Nav Mesh Obstacle, Box Collider with isTrigger set to false, Rigidbody with isKinematic set to true, *Stats* script, *Healthbar* script, a script that inherits from the *Building* script and which has the same name as the building, and *PlayerBuilding* script or a script that inherits from it.

As far as structure goes, they all have some renderable objects in their hierarchy which comprise the visual representation of the building in the game. Besides that, all the buildings have a MinimapIcon element which creates a 2D circle around the building through its Sprite Renderer component. This element's layer is set to 'Minimap', the result of which is that buildings are shown as coloured circles on the minimap.

All the buildings also have a Canvas element with one child called HealthSlider. This HealthSlider has a Slider component attached to it, which is used to display the amount of hit points (HP for short) that the building has left. When a building has no HP left, it is destroyed. The *Healthbar* script controls the behaviour of this Slider.

The *Stats* script contains data about different stats (properties) that the given building has, like amount of HP, armour, power, etc.

All building GameObjects have a 'Building' tag.

3.3.1. Farm

The Farm is used for generating resources. This property is ensured by its *Farm* script.

3.3.2. Tower

The Tower is an offensive building capable of shooting at enemies that are within a certain radius of it. To this end, it has an extra trigger Collider attached. It is a Sphere Collider and its purpose is the detection of enemy units and buildings in the Tower's attack radius. Once such an entity is detected, the tower starts attacking it. This functionality is implemented in the *Tower* script.

Due to the fact that this building has another Collider, I had to set the layer of its root element to 'Ignore Raycast'. This is because raycasting is used in this game whenever the player wants to click on something, and if the Tower's root element wasn't ignored by this operation, the ray would always be stopped by it. This is also the reason why the Tower has another trigger Collider on its Body element. The layer of this element is 'Default', so its Collider can be picked up by raycasting.

3.3.3. UnitCreationProcess

This object has a script with the same name attached to it. It is responsible for the creation of units. Every building capable of creating units instantiates one UnitCreationProcess for every type of unit that it can create.

The reason I made a separate GameObject for creating units is that I wanted each unit type to have its own creation queue. That means that units of the same type are created serially, while units of different types are created in parallel. To achieve this, each unit type had to have some kind of queue associated with it.

A straightforward solution would have been to hard-code the necessary number of queues into the script of each building. However, that would have made adding new unit types quite difficult. Therefore, I needed some kind of object that could be instantiated as many times as necessary and that, given a type of unit, would be able to manage the creation queue of that type.

To manage this queue, I needed some kind of way of creating a unit after a specific time interval. I tried to use the C# *Timer* class, but I couldn't get it to work because it deals with C# delegates and events. I had no experience with these concepts, so I opted for *MonoBehaviour* functions instead. However, *MonoBehaviour* or a class inheriting from it shouldn't be instantiated through the *new* keyword. Rather, it should be created as a part of a GameObject when it is instantiated. Because of this, I created the UnitCreationProcess GameObject.

3.3.4. Base & Barracks

The Base and the Barracks are very similar, since they are both buildings for creating units. The Base generates Workers, while the Barracks can generate fighter units. Besides all the things that they have in common as buildings, they share two more elements in their composition hierarchy – SpawnDestination and SpawnPoint. SpawnPoint determines where the created units are be instantiated and SpawnDestination where they head right after instantiation.

3.3.5. Outlines

Every building has a corresponding outline object. This object copies the shape of the building but retains almost none of its components. The purpose of these objects is twofold. First, they serve as placeholders for buildings that are to be built. When a Worker is going to build a building at some location, an outline of that building is

placed there first, so that the other Workers cannot put their buildings there. Second, these objects are used by the player to decide where to build a building.

These two objectives are accomplished thanks to two components attached to every outline's root element - a Box Collider and an *Outline* script. The Collider allows the outline to be detected by Workers, preventing them from trying to build a building there. The *Outline* script makes the outline turn red whenever the player places it in a spot where the corresponding building cannot be built. It also prevents the player from building on that spot.

3.3.6. BuildingPointer

An object only used by the player. It is a pointer which starts hovering above a building if the player selects it. It has an Animator component in its root element which ensures that it moves up and down.

3.3.7. Smoke

This GameObject serves as a placeholder for a building while it is being created. When the Worker starts building a building, this GameObject is instantiated and stays in the scene until either the building is finished or the object is destroyed.

Since it is supposed to be a stand-in for a building, it is similar to one in many ways. Its root element has a Rigidbody, Box Collider, *Stats* script, *Healthbar* script and *PlayerBuilding* script attached. Also, the *Smoke* script inherits from Building and one of the children of this GameObject's root element is a Canvas with a Slider for measuring HP.

The smoke effect itself is generated by a Particle System (a built in Unity effect).

During the building process, the Smoke's health slider fills up. It can be attacked and take damage during this process and if its HealthSlider hits zero, the building process is terminated. If it doesn't, the building isn't instantiated with a full HealthSlider. Instead, the amount of damage that the Smoke received is subtracted.

3.4. Units

Every unit GameObject has a 'Unit' tag.

3.4.1. Warrior, Archer & Mage

These are three types of units that an agent can create and use for offense. They have different abilities, but structurally they are almost the same. They consist of the following – a root element which supplies most of the functionality, a Camera, a Body, left and right hands (LHand and RHand respectively) with the right being equipped with an aiming Camera, a Canvas, Highlight, MinimapIcon, LevelCanvas,

and three empty GameObjects called BodyItem, TopItem and Trinket, which serve as holders for different kinds of items.

The Canvas and MinimapIcon fulfil the same functions as they do in buildings, but units also have a ManaSlider as one of the children of the Canvas.

The Highlight is just a translucent outline of the body of the unit. It serves to show the player when he/she has selected the unit.

The LevelCanvas is a Canvas on which the level of the unit is displayed when in 3rd person mode. It also has a slider which shows how much experience the unit has gained since levelling up. The important difference between this Canvas and the unit's other Canvas is that the LevelCanvas has its Render Mode set to Screen Space – Overlay, while the other one has it set to WorldSpace. This is so that the health- and manabars are supposed to move with their corresponding units, while the LevelSlider is supposed to be permanently stuck to the screen.

The two hands don't have any renderable components attached to them, they are just Transforms that signify where weapons should go. The Camera on the right hand is used for aiming by units with range weapons.

The root element has many components, including a Rigidbody and two Colliders (one for colliding and another, larger one, for the detection of enemies) and also the Nav Mesh Agent component. Furthermore, the root element has the following scripts – *Spellbar*, *Healthbar*, *Manabar*, *Stats*, *Equipment*, *SpellbarGUI*, *EquipmentGUI*, *PlayerControls* and *UnitAI*. Exactly one of the last two scripts is active at any time, depending on whether the unit belongs to the player or to an A.I.

One of the few structural differences between these units is that they each have a unique script attached, which is named after them.

The Body has three important components – a *Death* script, an Animator and a Collider. The *Death* script says what is supposed to happen when a unit dies. It has to be attached to the Body, not the root element. This is so, so that its *Die* function can be invoked from the death animation. For this to be possible, it has to be attached to the same element as the Animator component.

The Collider attached to the Body is a trigger Collider. It is used to detect when the player has clicked on the unit. It is not attached to the root element, because the layer of the root element is set to 'Ignore Raycast'. This is for the same reason as with the Tower.

3.4.2. Monster

A Monster unit is basically a watered-down version of the other fighter units. As far as structure goes, it is missing the Highlight, both hands, and the three item holders. The LevelCanvas and Camera components are present only for convenience, since the *AdvancedUnitFunctions* script, which all unique unit scripts inherit from, works with these elements.

As far as other unit scripts go, the *Spellbar* script is missing, as are the *PlayeControls* and *UnitAI* scripts, and both of the GUI scripts.

Simlarly to other units, the Monster also has a unique script which is named after it. Among other things, it ensures that, when not engaging an enemy, the Monster moves randomly within a certain radius from some given point.

3.4.3. Worker

A worker only consists of the root element, a Body, a MinimapIcon and a Canvas. The Body has the same components as the Bodies of other units, except for the fact that, instead of the *Death* script, it uses the *WorkerDeath* script. This is necessary because the death of a Worker is different in two aspects. First, no Loot Box (3.7.1.) is left behind. Second, a part of the Worker's Body is a Particle System, which I chose to stop via a script because it looks better than just destroying it.

The Worker's root element has a *Worker* script attached to it. This script allows it to build buildings. The root element also has two other scripts not used anywhere else attached – the *WorkerControls* script, which inherits from *PlayerControls*, and the *WorkerAI* script. Only one of these can be active at any given time.

3.5. Icons

Unlike all the previous entities that I've talked about, icons aren't GameObjects, just plain C# objects. They can be divided into 3 categories – spells, items and building/unit icons.

Every icon has a variable of type *Texture2D*. An image is stored in this variable, which can be shown to the player. The player can then activate this icon's functionality by left- and/or right-clicking the image.

3.5.1. Spell icons

These icons are used to trigger spells. Their corresponding images appear in the spell bar at the bottom of the screen when the player is playing in 3rd person mode. Each fighter unit, except for the Monster, has two unique spells. The Monster has no spells.

Archer's spells

Damaging Aura – Instantiates a GameObject called DamagingAuraSpell, which has a Light component and a Sphere Collider attached. It also has a script named after it attached. This script keeps damaging enemy units that have entered the Sphere Collider.

Teleport – Instantiates a GameObject called TeleportEffect and destroys it after 1 second. The GameObject has a *TeleportEffect* script attached which ensures that the unit is teleported to the nearest friendly building when the object is destroyed.

Mage's spells

Strength and Durability Buff – Finds nearby friendly units and temporarily increases their 'Armour' and 'Power' stats. It also creates a SpellTimer GameObject which stops the spell after a given amount of time.

The reason why I created this GameObject was similar to why I created the UnitCretionProcess GameObject – I needed to use one of the *MonoBehaviour* functions to stop spells after a given amount of time. However, icons do not inherit from *MonoBehaviour*, so I needed to create another object which did.

Heal – When the player left-clicks on this spell's icon, a HealSpellProxy object is created. This object has the *HealSpellProxy* script attached which ensures that when the player left-clicks on a unit which is on the same team, it gets healed. Right-clicking anywhere deactivates the spell. This object was created for the same reason as the SpellTimer object. While a HealSpellProxy object exists, a small plus sign is drawn next to the mouse cursor.

Warrior's spells

Rage – Temporarily increases the Warrior's 'Power' stat. Also creates a SpellTimer which stops the spell after 30 seconds.

Shield – Instantiates the Shield GameObject and increases the unit's 'Armour' stat.

3.5.2. Items

Items are icons that can be stored inside an inventory. These icons can also be stacked, that is, if there are two items with the same name, they aren't be stored separately. Instead, the counter in the top right corner of the icon's image is incremented.

There are five items in this game. The first two are potions – a health potion and a mana potion, which replenish some of the unit's health or mana respectively. The other three are the different weapons used by the three different kinds of units. They all have a GameObject of their corresponding weapon stored inside them. That GameObject is instantiated when the weapon is equipped.

3.5.3. Building/Unit Icons

These icons serve to create or destroy buildings and units. They are always shown in the menu, just like the inventories of units. When it comes to Unit Icons, they are displayed when the player either clicks on a Base, or a Barracks and left-clicking them causes a unit to start being created, while right-clicking them causes one to stop being created.

The building icons, on the other hand, are displayed when the player clicks on a Worker. When one of them is left-clicked, an Outline of the building appears

and when the player left-clicks somewhere on the Terrain where the building can be built, the Worker heads over there and start building.

Lastly, there is the Destruction Icon. Every building has this Icon and by left-clicking it, the user can destroy the building and receive back 1/3 of the amount of resources used to build it.

3.6. Weapons

There are three weapons in this game – one for each type of unit. The Warrior uses a melee weapon called Basic Sword. The Archer uses a range weapon called Basic Bow, which shoots Arrow GameObjects. Lastly, the Mage uses a mid-range weapon called Basic Staff, which shoots MagicBall GameObjects.

The Basic Sword and both kinds of projectiles have the *WeaponEnd* script attached to some element of their composition hierarchy. This script is necessary to allow fighting in 3rd person mode.

3.7. Other GameObjects

3.7.1. Loot Boxes

Every unit besides the Worker has one loot box associated with it. The structure is always the same and quite simple – an empty root element, a Body, and a Particle System. The root element is present because I wanted the bodies of the Loot Boxes to spin, while the Particle Systems stayed stationary.

The body has three scripts – *LootBox*, *ItemDatabase* and *ItemDatabaseGUI*. The first just takes care of the animation and of destroying the Box after 30 seconds. The second takes care of storing items in the Loot Box. The last allows the player to display the items in a Loot Box and allows manipulation with them.

The Particle System is present just for decoration.

3.7.2. CoreController

Every agent has one CoreController associated with it. It stores the strategic functions of the agent in two scripts. The first is called *StrategicData*. It stores important data about the agent, like the number of units it currently has. It also has all the general functions that both an A.I. and the human player are supposed to have, like checking if there are enough resources to do some action. The other script is either the *PlayerStratControls* script or the *AIMain* script. The former has all the functions that the player uses to manipulate and interact with his units and buildings. The latter is the core of the basic A.I. that I have implemented in this game.

3.7.3. PlayerCanvas

This GameObject is instantiated from within the *PlayerStratControls* script and it is comprised of a root element and three children. The root element contains the Canvas component. The first of its children is called Minimap. This element contains a component called Raw Image which has a texture onto which the field of view of the MinimapCamera is projected.

The second is the Resources element which contains a Text component and an Outline component (not to be confused with the *Outline* script that the outlines of buildings have). It uses these components to display the amount of resources that the player currently has.

The third is called Notifications. Similarly to Resources, it also has the Text and Outline components attached. Its purpose is to allow notifications like “Insufficient resources” to be shown in the middle of the screen.

4. Developer documentation

In this chapter, I go over most of the scripts used in this game and explain what each of them does. I start with the *GameMaster* script which is in a category of its own because it takes care of setting up the game. I then explain some basic scripts that are used by various GameObjects and that are necessary to understand the parts that follow. After that I divide the scripts into these categories: buildings, units, A.I. scripts, player scripts, icons, weapons, spells and other scripts, and go over each of them individually.

I chose to describe them in this order because I think it best illustrates the structure of my game. I explain the workings of the basic building blocks of this game – buildings and units – after explaining some scripts that are necessary to both of these categories. I then explain how the A.I. and the player control these entities and then go on to explain additional parts of this game.

Before I do any of that however, I need to explain some Unity classes and some of the functions they offer.

4.1. Unity functions

4.1.1. MonoBehaviour message functions

The *MonoBehaviour* class is a class from which every script that is to be used as a component must inherit. Therefore, unless stated otherwise, assume that any class mentioned in the following sections inherits from *MonoBehaviour*.

This class allows the user to use functions that are a part of the Unity life cycle – a loop which goes over some of the functions offered by *MonoBehaviour* in a specific order every frame of the game. There are a lot of these functions, so I only mention the ones that I've used in my implementation here. An illustration of the life cycle can be found in the Unity user manual [6].

void Awake()

This is the first function called in the Unity life cycle. It is called when the script instance is being loaded and is usually used to initialize variables. It is automatically called once for every script, though that doesn't stop the user from also calling it through different scripts.

void Start()

Start is also only called once for every script, unless called additionally by the user. It is called on the frame when the script is enabled for the first time just before any methods concerned with updating but after the *Awake* function. The initialization of

different scripts can be interdependent, so knowing that one initialization function is definitely called before/after the other can be useful.

void OnTriggerEnter(Collider other)

This function is triggered by a Collider intersecting with another Collider. It allows the user to define ways of reacting to these intersections. For this method to be called, two conditions must be met. First, the GameObject to which a script with this method is attached must also have a trigger Collider attached. And second, at least one of the colliding GameObjects has to also have a Rigidbody attached.

The parameter passed to this function is the Collider which the GameObject's trigger Collider has intersected with.

void OnTriggerExit(Collider other)

Just like the previous function, only it gets called when two Colliders stop intersecting each other.

void Update()

This function gets called once every frame. It allows updating of the game state, which makes it essential for any game.

void OnGUI()

Used for rendering and handling GUI events (like user input or events that have something to do with rendering). It can be called multiple times per frame, once for every event.

void OnDestroy(), void OnDisable(), void OnEnable()

Get called when the script in which they are implemented gets destroyed, disabled, or enabled respectively.

4.1.2. Other MonoBehaviour functions

Some of these functions are implemented in one of the classes from which *MonoBehaviour* inherits, which means that they are present in other classes as well. For example, the *GetComponent* function is implemented in the *Component* class, from which *Transform* also inherits. I explain these functions only once though, since they always do the same thing. If, at some point, I call a *MonoBehaviour* method from a class that doesn't inherit from *MonoBehaviour*, assume that the method is implemented in some common ancestor of those two classes.

void Invoke(string methodName, float time)

Invokes a function. *methodName* specifies the name of the function to be invoked, while *time* specifies the time delay after which it should be invoked.

void InvokeRepeating(string methodName, float time, float repeatRate)

Invokes a function repeatedly. The first two arguments are the same as in *Invoke* and fulfil the same function, while the third determines the time delay between repeated callings of the invoked method.

static void Destroy(Object obj, float t)

Destroys a given Object after *t* seconds. This can be a GameObject, component, or asset. *t* is 0 by default.

static void Instantiate(Object original, Transform parent)

Takes a component, asset or GameObject as an argument and returns its clone. This clone's parent is defined by the second argument. The string "(Clone)" is added to the clone's name.

static void Instantiate(Object original, Vector3 position, Quaternion rotation)

This overload of the *Instantiate* method works in much the same way as the previous one. The difference is that it doesn't set the clone's parent and that it uses the *position* and *rotation* arguments to set its position and rotation.

T GetComponent()

Probably the most essential method besides the message methods. It returns a component of type *T* which is attached to the GameObject, or null if there is no such component.

Besides this function, there also exists a *GetComponents* function, which returns an array of all attached components of the given type. There are also variations of both of these methods which search not just the element to which they are attached, but also all of its children using depth-first search. They are called *GetComponentInChildren* and *GetComponentsInChildren*.

Coroutine

A coroutine is not a function, rather, it is a kind of function. Normal functions happen within one frame, which makes it impossible to use them to implement gradual changes, unless they are coupled with some of the Unity message functions that I've mentioned.

A coroutine is basically a function that has the ability to pause execution but then continue where it left off later. It is declared with a return type of *IEnumerator* and with the 'yield return' statement included somewhere in the body. That statement is the point at which execution pauses and resumes later.

Coroutine StartCoroutine(IEnumerator routine)

This overload of the *StartCoroutine* function can be used to start a coroutine by passing the function a variable of type *IEnumerator*, which corresponds to a coroutine.

Coroutine StartCoroutine(string methodName, object value)

This overload can be used to start a coroutine by passing it the name of the coroutine and passing its argument in *value*. An obvious downside of this approach is that the coroutine to be started has to take only one argument. Also, this version has a higher runtime overhead than the previous version.

void StopCoroutine(IEnumerator routine)

This method stops the coroutine stored in *routine*. Should only be used to stop a coroutine which was started by the overload of *StartCoroutine* which takes a parameter of type *IEnumerator*.

void StopCoroutine(string methodName)

Stops a coroutine called *methodName*. Should only be used to stop a coroutine which was started by the overload of *StartCoroutine* which takes a parameter of type *string*.

4.1.3. Resources

static T Load(string path)

Allows the user to load assets like prefabs (a predefined GameObject stored as an asset) or images. For this function to work, a folder called Resources has to be created, within which the assets are stored.

4.1.4. Transform

Transform Find(string n)

Finds a child named *n* and returns it, or returns null if no such child is found. Unlike *GetComponentInChildren* though, this function searches only the immediate children of the element to which it is attached.

Transform GetChild(int index)

Returns a child of the Transform at the given index or generates a "Transform child out of bounds" error, if the index is too large.

void LookAt(Transform target)

Rotates the GameObject so that it points towards *target*.

void Rotate(Vector3 eulers, Space relativeTo)

Rotates the GameObject around the x, y and z axes according to the angles given in *eulers* (a variable of type *Vector3*, which is a 3D vector). The second argument determines whether the GameObject should be rotated around its own axes or the world's axes. By default, the GameObject's own axes are used.

void Rotate(float xAngle, float yAngle, float zAngle, Space relativeTo)

Works the same as the previous overload, only it takes the amount by which the object should be rotated around each axis from the first three parameters instead of a vector.

void Translate(Vector3 translation, Space relativeTo)

Moves the GameObject in the direction and by the distance given by *translation*. Just like with the *Rotate* function, the movement can be applied in relation to the world's axes, or the GameObject's axes, depending on the second argument.

4.1.5. GUI

This class provides various functions for drawing GUI elements like boxes or textures. Its methods often use the *Event* class, which contains information about GUI events like user input.

4.1.6. Input

Provides an interface for reading input data.

4.2. GameMaster

This is the script that starts running when the application is started. It takes care of setting up the game and it stores all the information necessary to do so. It provides static data and methods that should be accessible to all agents at all times. The functions which are called by the UI elements in each menu are also implemented in this script.

Important Variables:

bool playing – If this boolean is true, the game is in progress.

List<StrategicData> agents – A list of *StrategicData* (4.3.5.) of all the agents.

List<GameObject> exceptions – A list of *GameObject*s that aren't supposed to be deleted when the game is restarted.

Important Functions:

void Start() – Initializes all the variables and ensures that, out of all the menus, only the Main menu remains active.

void Play() – The Play button in the Main menu calls this function. It starts up the game. First, it creates 2 agents in opposing corners of the map, one A.I. and one human. Then it creates two batches of 5 monsters each in the remaining two corners. Finally, it deactivates the Main menu and sets the *playing* boolean to true, indicating that the game has started.

void GameOver() – Checks whether the game has ended. It does so by cycling through *agents* and checking whether they still have any buildings or units left.

void Update() – Takes care of pausing the game when the player presses the escape key. Also keeps checking whether the game hasn't ended using the *GameOver* function. If it has, this function stops time in the game and displays the Game Over menu.

static bool CameraOnTerrain() – Checks whether the strategic Camera is within the bounds of the Terrain and whether it isn't too close.

4.3. Basic scripts

This section goes over the scripts which do not belong in any of the other categories, but are often necessary to understand the scripts in them.

4.3.1. Stats

Contains an entity's stats – numerical values that represent an entity's properties, like power or amount of health.

Important Variables:

List<BaseStat> statsList – Stores all the stats of the given entity as *BaseStat*s (4.3.2.).

4.3.2. BaseStat

Stores the value of one stat of some entity.

Important Variables:

List<StatBonus> baseAdditives – Stores all the bonuses added to this stat. The *StatBonus* type is explained next.

int baseValue – The initial value of the stat.

string statName – The name of the stat.

float modifier – The value by which the stat is supposed to be multiplied when the entity levels up.

Important Functions:

int GetCalculatedStatValue() – Returns a sum of *baseValue* and the values of all the bonuses in *baseAdditives*.

4.3.3. StatBonus

Contains a bonus that is to be added to some stat.

4.3.4. PlayerOrAISetup

This script is used to set up whether the given entity belongs to the player or to an A.I.

Important Functions:

bool PlayerOrAI() – Returns true if the entity belongs to a player.

void SetPlayerOrAI(bool flag) - Either activates all of the given GameObject's player scripts and deactivate its AI scripts, or vice versa.

4.3.5. StrategicData

Inherits from *PlayerOrAISetup*. It is attached to the CoreController GameObject (3.7.2.). It holds information about the agent, manages its resources and also takes care of setting up the agent's base at the beginning of the game.

Important Variables:

int team – The team to which this agent belongs.

int player – The agent's id.

int resources – The amount of resources that the agent has.

List<GameObject> buildings – A list of all the buildings belonging to this agent.

List<GameObject> units – A list of all the units belonging to this agent.

Important Functions:

void SetUpBase() – Sets up the agent's base at the beginning of the game.

bool CheckResources(int amount) – Checks whether there is enough resources for some task.

4.3.6. IDamageObserver

An interface implemented by classes that need to be notified when an entity receives damage.

4.3.7. CommonFunctions

Inherits from *PlayerOrAISetup*. It is an abstract class which contains the common functionality used by both units and buildings.

Important Variables:

GameObject coreController – Stores the CoreControler GameObject (3.7.2.) of the agent to which the entity belongs.

int team – An id of the team to which the entity belongs.

int player – An id of the player to which the entity belongs.

List<IDamageObserver> observers – A list of objects that are supposed to be notified when the entity takes damage.

Important Functions:

void ReceiveDamage(int dmg, GameObject attacker) – Drains the corresponding amount of the units health and notifies all the objects in *observers*. The amount of drained health doesn't have to be equal to *dmg* though – an entities 'Armour' stat can decrease it. If the entity's healthbar is fully drained by receiving *dmg* and *attacker* is a unit, *attacker* receives some experience. The amount is defined by the entities 'exp' stat.

abstract void SetColor(Color c) – Used to colour the GameObject.

abstract void Terminate() – Used to define what should happen when the entity is to be killed/destroyed.

virtual void SetStats() – Sets the entities stats in *Stats* script. The reason this method is virtual and not abstract is that the stats of the entity can only be set from its unique script. Therefore it would just be left empty in the classes that inherit directly from *CommonFunctions*.

4.3.8. Valuebar

An abstract class that controls some kind of slider.

Important Variables:

Slider slider – The slider which this script controls.

string statName – The name of the stat which determines how fast *slider* should regenerate itself.

Important Functions:

void Update() – Updates *slider*'s value and keeps it turned towards the camera.

void Regenerate() – Increments the value of the slider by 1. Is called via *InvokeRepeating* in *Start* with a period that is computed based on the entities stat *statName*.

bool Empty() – Returns true if the slider's value is 0.

4.3.9. Healthbar

Inherits from *Valuebar*, controls an entity's HealthSlider element.

4.4. Buildings

4.4.1. BuildingType

An enum of the different kinds of buildings.

4.4.2. Building

Inherits from *CommonFunctions*. Every building has a script which inherits from this one attached to it.

Important Variables:

BuildingType type – Stores the type of the building.

4.4.3. Farm

Inherits from *Building*.

Important Functions:

void Update() – Keeps the Farm's Canvas element turned towards the camera.

void MakeResources() – Creates 50 resource units and displays that number in a text above the Farm. It is called every 20 seconds via *InvokeRepeating* in the *Start* method.

4.4.4. Tower

Inherits from *Building*.

Important Variables:

List<GameObject> targets – A list of enemies that are within the tower's range.

Important Functions:

void OnTriggerEnter(Collider other) – If an enemy enters the Tower's trigger Collider, it is added to *targets*.

void OnTriggerExit(Collider other) – Removes the GameObject that has exited the trigger from *targets*.

void Attack() – Chooses a random target from *targets* and shoots at it. It is called in the *Start* function via *InvokeRepeating* with a 2 second period.

4.4.5. UnitCreationProcess

A script responsible for creating units.

Important Variables:

GameObject unitType – A prefab of the type of unit that this *UnitCreationProcess* can create.

int creationTime – The time it takes to create the given unit type.

IIcon icon – An icon associated with the given type of unit. The *IIcon* type is described at (4.8.1).

Important Functions:

IEnumerator MakeUnit() – A coroutine that makes a new unit every *creationTime* seconds, while the *count* variable in *icon* isn't 0.

void AddUnit() – Increments the *count* variable in *icon*. If the *MakeUnit* coroutine currently isn't in progress, this function starts it.

void StopCreating() – Decrements *icon.count* and returns the resources allocated for creating the unit.

4.4.6. UnitCreatorBuilding

Inherits from *Building*. Any building that is capable of creating units has a script that inherits from this script attached.

Important Variables:

Dictionary<UnitType, int> unitCosts – Stores the costs of all the *UnitType*s (4.5.1.) which this building can create.

Dictionary<UnitType, UnitCreationProcess> unitCreationProcesses – A dictionary which associates the different *UnitCreationProcess*es with their *UnitType*s.

Important Functions:

void AddUnitType(UnitType id, GameObject go, int index, string name, string description, GameObject building) – Adds a new type of unit that the building can create and creates its icon.

void CreateUnit(UnitType id) – Creates a unit of the given type if there are enough resources for it.

4.4.7. Barracks

Inherits from *UnitCreatorBuilding*. Sets the stats of the building in the *Awake* function and sets the building's type in the *Start* function. The *Start* function also defines the types of units that it can create.

4.4.8. Base

Just like the *Barracks* script, it is an overlay of *UnitCreatorBuilding*. It provides the settings that enable the base to fulfil its functions.

4.4.9. Outline

Important Variables:

int triggerCounter – Stores the number of intrusions into the outline's Collider.

bool buildable – Indicates whether the corresponding building can be built at the place where the outline is currently positioned.

Important Functions:

void OnTriggerEnter(Collider other) – This function colours the outline red when its Collider intersects with the Collider of a building or that of another outline. It also increments *triggerCounter*.

void OnTriggerExit(Collider other) – If the outline's Collider stops intersecting with another Collider which belongs to either a building or another outline, *triggerCounter* is decremented. If its value is zero, that means that there are no more objects preventing the corresponding building from being built on that spot. As a result, the outline is coloured green and the *buildable* boolean is set to true.

4.4.10. Smoke

Inherits from *Building* and implements the *IDamageObserver* interface.

Important Variables:

int amount – The amount of health that is to be replenished in every iteration of the *HealthFillUp* coroutine.

GameObject worker – The Worker working on the building for which the Smoke object is a placeholder.

Important Functions:

void Update() – Keeps changing the alpha value of the MinimapIcon element's colour.

IEnumerator HealthFillUp(float t) – Fills up the Smoke's healthbar by adding *amount* of health to it in every iteration. When it finishes filling it up, it calls the Worker's *FinishBuilding* function.

void OnDestroy() – Reactivates the Worker upon being destroyed. This is only necessary for the case when the Smoke is destroyed by enemies, not when the building is finished. In that case, the Worker is reactivated before the Smoke is destroyed.

4.5. Units

4.5.1. UnitType

An enum of the different types of units. The fighter units creatable by an agent have been assigned prime numbers which provides an easy way of checking whether an item can be equipped by a given unit. Multiply the numbers of units that the item can be equipped by and then divide by the number of the unit which wants to equip the item. If the remainder is 0, then the item can be equipped by the given unit.

4.5.2. IEngage

An interface which declares the *Engage* function which is used to define how a unit should behave when an enemy comes within a certain distance of it.

4.5.3. BasicUnitFunctions

A set of functions that every unit must have. It inherits from *CommonFunctions*.

Important Variables:

Transform target – The Transform of a unit with which the unit is engaged in battle.

bool onMyWay – If this boolean is true, it means that the unit should ignore all enemies until it gets to its destination.

Important Functions:

override void Terminate() – Deselects the unit, removes it from CoreController's (3.7.2.) list of units and triggers the unit's death animation.

void GoToPoint(Vector3 p) – Sends the unit to the given point. To do this, it just sets the destination of the NavMeshAgent component which then takes care of pathfinding. In strategic mode, units are navigated exclusively through this function.

void OnTriggerEnter(Collider other) – If an enemy enters the unit's trigger Collider, the unit tries to engage it using the *Engage* function of a component which implements the *IEngage* interface.

Transform FindClosestEnemyInRadius(float radius) – This function finds all the enemy units and buildings in a given radius. It does so by using the *Physics.OverlapSphere* function which takes two arguments – a point and a radius. It then finds all the *Collider*s within the sphere defined by those parameters. It is a function supplied by Unity.

float TrueDistance(Transform other) – Returns the distance between the centre of the GameObject and the closest point on *other*'s non-trigger Collider. This is necessary because if a unit were to measure the distance between it and its target based on the distances between their centres, it might never get 'close enough' to a building to attack it.

4.5.4. Worker

Inherits from *BasicUnitFunctions*. Also implements the *IEngage* interface, but leaves the *Engage* function empty. This is because the Worker has no means of attacking.

Important Variables:

GameObject toBuild – Stores a prefab of the building that is to be built.

GameObject smoke – Stores the Smoke GameObject.

bool buildingInitiated – If this boolean is true, the Worker has been sent to create a building.

Important Functions:

void SetBuildingSite(Vector3 point) – Sets the place where a building is to be built and sends the Worker there. Also sets *buildingInitiated* to true.

void Build() – This method starts the building process. After checking whether there are enough resources to build a certain building, it instantiates a Smoke GameObject and sets all of its settings. It then subtracts the amount of resources necessary to create the given building and deactivates the Worker.

void FinishBuilding() – Finishes the building process by instantiating the building stored in *toBuild*, setting all of its necessary settings, destroying *smoke* and activating the Worker. It is called from within the *Smoke* script.

4.5.5. AdvancedUnitFunctions

Inherits from *BasicUnitFunctions* and implements the *IDamageObserver* and *IEngage* interfaces. Every unit capable of fighting has a script which inherits from this script attached to it.

Important Variables:

int experience – The amount of experience that the unit has gained since last levelling up.

float aggroRadius – The radius within which the unit registers and attacks enemies.

float fleeRadius – If the unit is engaged in a battle, but the distance between it and its target becomes larger than fleeRadius, the unit stops targeting the other unit.

float attackDistance – The minimum distance that has to be between a unit and its target for it to be able to attack.

Important Functions:

void Update() – Takes care of deciding how a unit should behave if it has a target. Makes the unit always turn towards the target. If the target is within *attackDistance*, the unit starts attacking. If it is within *fleeRadius*, it tries to follow it. If it is further than *fleeRadius* and it isn't a building, the unit stops targeting it.

void DealDamage() – If the unit has a target and a weapon, this function gets the units 'Power' stat and passes it to the target's *ReceiveDamage* function along with the *GameObject* of the unit.

IEnumerator AttackEnumerator(float t) – This method is called as a coroutine. It makes sure that the unit deals damage to its target if it is close enough by calling the *DealDamage* function. It also fires the attack animation. If the unit loses its target, this function tries to find another enemy within *aggroRadius* using the *FindClosestEnemyInRadius* function. If one is found, the unit engages it. Otherwise, the coroutine ends.

void Attack() – Starts the *AttackEnumerator* coroutine with a period of 2 seconds.

void Engage(Transform other) – If the unit has no target, hasn't been commanded to go somewhere else (*onMyWay* is false), and *other* is an enemy, the unit sets *other* as its target and goes to attack it.

void DamageReceived(int amount, GameObject attacker) – If the unit's 3rd person mode isn't currently being used, this calls the *Engage* function on *attacker*'s *Transform*.

void AddExp(int amount) – Increases the unit's *experience* by amount, which is reflected by its level slider. Also causes the unit to level up when *experience* reaches the necessary value.

4.5.6. Archer, Mage & Warrior

These scripts inherit from *AdvancedUnitFunctions*. They set up all of the given unit type's unique settings.

4.5.7. Monster

Inherits from *AdvancedUnitFunctions*. Like the other fighter unit scripts, it sets up the unit's settings. Besides that, it also ensures that the Monster moves around randomly within a certain area.

Important Variables:

Vector3 startingPoint – The point which is at the centre of the area in which the Monster can randomly move.

float distance – The maximum distance that the Monster can go from *startingPoint* in both x and z directions (separately – this makes the are within which the Monster can move a square). If the Monster is attacked, this restriction no longer applies.

Important Functions:

void Update() – Once the Monster's *Nav Mesh Agent* has stopped moving, this function invokes the *Move* function with a 1 second delay.

void Move() – Chooses a random point which is at most *distance* away from *startingPoint* in both x and z directions and sends the Monster there.

4.5.8. Death

This script is attached to the body element of every fighter unit. It ensures that the unit leaves behind a Loot Box.

Important Variables:

GameObject lootBox – Stores the prefab of the Loot Box associated with the given unit.

Important Functions:

void TransferItems(ItemDatabase lootBox) – Removes all the items from the units inventory and equipment and stores them in the Loot Box. The *ItemDatabase* type is described at (4.11.2.).

void Die() – Instantiates the Loot Box, colours it the same colour as the unit had and transfers the unit's items to it using the *TransferItems* function. It is called from within the unit's death animation.

4.5.9. WorkerDeath

The Worker has to have a different script for dying than other units because it doesn't leave behind a loot box. The functions of this script are called from within the Worker's death animation.

Important Functions:

void Die() – Simply destroys the Worker.

4.5.10. Equipment

A script that all fighter units have. It stores and manages their equipment.

Important Variables:

List<IEquippable> parts – A list of all the equipped icons. The IEquippale type is described at (4.8.7.).

Dictionary<int, Transform> itemHolders – A *Dictionary* which binds together the numbers of body parts with the *Transform*s used for holding them.

Dictionary<int, GameObject> equippedItems – A *Dictionary* which binds together the numbers of body parts with the *GameObject*s of items assigned to them.

Important Functions:

void Equip(IEquippable toEquip) – Equips an item. If an item has already been equipped to the body part to which *toEquip* can be equipped, it is unequipped.

void Attack() – In the case of the Monster, its attack animation is triggered. In the case of other units, this function triggers the attack animations of the weapons that the unit has equipped in its hands.

4.5.11. Manabar

Inherits from *Valuebar*. Controls a unit's mana slider.

4.5.12. Spellbar

Stores the unit's spell icons.

4.6. A.I.

A.I. for games can range from incredibly simplistic binary decision-making agents to complex pieces of software that can rival the human mind. The best A.I. have to be able to learn and respond quickly to a wide variety of scenarios far greater than they

could hope to store in their memory. However, seeing as the purpose of my thesis wasn't to create a ground-breaking A.I., I opted for a more simplistic state-based approach.

I first wanted to implement this concept using a finite-state machine. However, when I tried to plan what states the game should have, I realised that this was more complex than necessary for my game.

Instead of a finite-state machine, every A.I. script in this game, with the exceptions of *AI* and *UnitAI*, has a coroutine called *ActionCycle*. This coroutine repeats itself after a set number of seconds. The period at which is does so is mostly arbitrary and differs between scripts. It checks some conditions and chooses how to act based on them. The *UnitAI* script uses the Update function for this purpose.

The A.I. in this game doesn't use spells. This is so for two reasons. The first is that it would be quite hard to come up with conditions under which each of the six spells implemented in this game should be cast. The second is that it would give it an unfair advantage over the player, who couldn't cast spells on more than one unit at once. I therefore decided to not let the A.I. use spells and make them available to the player only in 3rd person mode.

4.6.1. AI

An empty class that every AI script inherits from, so that it can be disabled/enabled via the *SetPlayerOrAI* function in *PlayerOrAISetup*. I decided to make it a class instead of an interface so that functions could be added to it in case of expansion.

4.6.2. AIMain

This script is attached to the CoreController GameObject (3.7.2.) and controls the A.I. units as well as providing some functions that other A.I. components use.

Important Variables:

Vector3 centre – The centre of a circle within which the Workers can build buildings.

float radius – The radius of the circle.

Important Functions:

IEnumerator ActionCycle – A coroutine which is started in the *Start* function with a period of 15 seconds. It monitors the number of units that the A.I. has that aren't currently attacking the enemy. If there are at least 5, it sends them to attack.

4.6.3. BaseAI

A script that controls the Base building.

Important Functions:

IEnumerator ActionCycle(float t) – This coroutine keeps checking whether the agent owns three workers. If not, it creates one.

void Start() – Starts the *ActionCycle* coroutine with a period of 11 seconds. This period isn't quite arbitrary – I needed to make it longer than the building time of a Worker (10 seconds), otherwise it would run again before the Worker finished being created and would start creating another one.

4.6.4. BarracksAI

Important Functions:

IEnumerator ActionCycle(float t) – A coroutine that runs every 5 seconds. It decides what unit should be created next based on the ratios between the already created types of units. It tries to keep the number of existing units of all types equal.

4.6.5. WorkerAI

Important Variables:

GameObject outline – Stores an outline of the building that the Worker is going to build.

Important Functions:

IEnumerator ActionCycle(float t) – Looks at the number of Farms and Barracks and, based on that, decides what building the Worker should build next. It tries to keep the ratio between these two types of buildings at 5 Farms to 1 Barracks.

void OnEnable() – Starts the ActionCycle coroutine with a period of 5 seconds. This happens every time the Worker is activated after finishing a building.

void OnDisable() – When this script is disabled, that means that the Worker has either died or started building. In both cases, the outline that the Worker has created needs to be destroyed which is what this function does.

void Build(string type) – Loads the building given by the *type* parameter from resources, then checks whether there is enough resources to build it. If so, tries to find a location where it can be built. If one is found, an outline of the building is instantiated there and the Worker is sent there to build the building.

4.6.6. UnitAI

A script used by every fighter unit belonging to an A.I.

Important Variables:

bool patrol – If this boolean is true, then the unit should keep moving around the A.I. camp. If it is false, it should attack the enemy.

Important Functions:

void Update() – Either keeps the unit moving from place to place within the camp or sends it to attack the nearest enemy unit, depending on the value of the *patrol* variable. Also makes it use its health potion, if its health is low.

4.7. Player

The scripts in this category provide the player with means of interacting with the game environment and issuing commands to units and buildings.

4.7.1. Player

A class which every script that is only used by the player inherits from, so that it can be disabled/enabled by the *PlayerOrAISetup* script. Unlike the *AI* script, this script isn't empty. It contains a *Start* function and some functions for creating GUI.

4.7.2. PlayerStratControls

Allows the player to control units and buildings in strategic mode as well as move the camera.

Important Variables:

List<IIcon> slots – The menu slots which can be filled up with icons of different entities. The *IIcon* type is described at (4.8.1.).

Important Functions:

void Update() – Calls the *LeftMouseButtonDown*, *LeftMouseButtonUp* and *CameraMovementControl* functions. Also ensures that pressing tab performs a switch between the strategic camera and a unit's camera, and that right-clicking somewhere on the map sends the selected units there.

void OnGUI() – Takes care of drawing all the 2D GUI elements.

void FillSlots(bool flag, List<IIcon> icons) – Fills the slots in the menu with the entities icons.

Vector2 MinimapToWorld(Vector2 pos) – Converts minimap coordinates to world coordinates.

void CameraMovementControl() – Allows the player to control the movement of the Camera.

void LeftMouseButtonDown() – Reacts to the left mouse button being clicked.

void LeftMouseButtonUp() – Reacts to the player letting go of the left mouse button.

4.7.3. PlayerBuilding

A script used by every building which belongs to the player.

Important Variables:

GameObject pointer – A prefab of the BuildingPointer (3.3.6.).

bool selected – If this boolean is true, that means that the given building is currently selected by the player.

Important Functions:

virtual void Select() – Sets *selected* to true, activates *pointer* and fills the menu slots with this building's icons.

4.7.4. PlayerUnitCreatorBuilding

Inherits from *PlayerBuilding*.

Important Functions:

void Update() – If the building is selected, this function places the SpawnDestination element on any place on the map where the player right-clicks.

override void Select() – Calls *PlayerBuilding*'s *Select* function but then also activates/deactivates the SpawnDestination element.

4.7.5. PlayerControls

A script that allows the player to control a given unit.

Important Functions:

void ThirdPersonControls() – Implements all the ways that a player can control the unit in 3rd person mode.

void Update() – If the player is playing for this unit in 3rd person mode and its healthbar drains completely, this function performs the switch to strategic mode. In 3rd person mode, it also calls the *ThirdPersonControls* function. If strategic mode is

currently being used, this method ensures that when the player right-clicks somewhere while the unit is selected, it is sent to that spot.

void StrategicCamSwitch(bool flag) – Performs the switch between the strategic camera and the unit's main camera. If *flag* is true, the switch is from strategic to third person. It also takes the MinimapLight element (mentioned at 3.2.6.) of the currently used camera and attaches it to the camera to which the switch is being performed.

void AimSwitch(bool flag) – This function switches between the unit's main camera and aim camera. If *flag* is true, the switch is to the aim camera.

virtual void GoToDestination(Ray ray) – Sends the unit to the destination which is pointed at by *ray*.

4.7.6. WorkerControls

Inherits from *PlayerControls*. Takes care of Instantiating the building outline when the player wants to build a building and of providing the player with ways of interacting with it.

Important Variables:

GameObject outline – The outline of the building that is to be built.

Important Functions:

void Update() – Allows the user to place *outline* somewhere on the map or to destroy it by right-clicking anywhere.

void GoToDestination(Ray ray) – This function overrides the one declared in *PlayerControls*, adding a condition. The Worker is only sent somewhere if it doesn't currently have an outline instantiated.

void OnDisable() – Destroys the outline of the building that is to be built. This ensures that the outline is destroyed both when the Worker starts building and when it is killed.

4.7.7. UnitCamera

Allows the player to control a unit's main camera.

4.7.8. AimCamera

Allows the player to control a unit's aim camera.

Important Variables:

Vector3 mouseDelta – The difference between the position of the mouse when *Update* is called and its position when *Update* was previously called.

Important Functions:

void Update() – Uses the *mouseDelta* value to move the aim camera in the direction in which the player moved his/her mouse.

4.7.9. ItemDatabaseGUI

Implements the GUI through which the player interacts with loot boxes.

Important Functions:

void Update() – Computes whether the database should be shown.

void OnGUI() – Draws the database as well as the descriptions of the different items in it.

4.7.10. EquipmentGUI

Provides GUI through which the player can interact with a unit's equipment.

4.7.11. SpellbarGUI

Provides the GUI through which the player interacts with the spellbar.

4.8. Icons

Icons are all plain C# objects, they don't inherit from *MonoBehaviour*. They are an important part of the GUI. They allow the user to interact with spells, items, or issue commands to units and buildings.

4.8.1. IIcon

An interface which every icon implements. It declares all of the important properties that every icon must have.

Important Variables:

Texture2D icon – The image which represents the icon to the player.

Important Functions:

void Click(bool button) – If *button* is true, it was left-clicked, otherwise, it was right-clicked.

4.8.2. BuildingIcon

An icon that can be used to create a building.

Important Functions:

void Click(bool button) – If *button* is true, an outline of the corresponding building as well as the building itself are loaded via *Resources.Load* and passed to the Worker.

4.8.3. DestructionIcon

An icon that can be used to destroy a building.

Important Functions:

void Click(bool button) – If *button* is true, the building is destroyed and 1/3 of the resources used to build it are returned to the player.

4.8.4. UnitIcon

An icon that can be used to create a unit.

Important Variables:

UnitType id – The type of unit that can be created by this icon.

Important Functions:

void Click(bool button) – If *button* is true, a unit starts being created. Otherwise, a unit stops being created.

4.8.5. PotionIcon

An abstract class. It represents an icon of a potion that can somehow be applied.

Important Functions:

void Click(bool button) – If *button* is true (the icon was left-clicked), the potion is applied. Otherwise, the potion is destroyed.

abstract void ApplyPotion() – Determines what applying the potion should mean.

4.8.6. HealthPotionIcon & ManaPotionIcon

Both inherit from *PotionIcon*. The former replenishes health while the latter replenishes mana.

4.8.7. IEquippable

An interface that inherits from *IIcon* and that is implemented by any equippable item's icon.

4.8.8. WeaponIcon

An abstract class which implements *IEquippable*. It is an icon which corresponds to an equippable weapon. Every weapon in this game has an icon associated with it which inherits from this class.

Important Variables:

bool equipped – This variable determines whether the item represented by this icon has been equipped.

Important Functions:

void Click(bool button) – If *button* is true and *equipped* is false, the item is equipped and removed from the inventory. If *button* is false and *equipped* is false, the item is destroyed. If *button* is false and *equipped* is true, the item is unequipped and placed inside the unit's inventory.

4.8.9. ISpellIcon

An interface which inherits from *IIcon* Most importantly, it declares the *CastSpell* and *StopSpell* methods. Every spell in this game has an icon associated with it which implements this interface.

4.9. Weapons

4.9.1. IWeapon

An interface which all weapons implement.

Important Functions:

void Attack() – Starts the attack animation.

void StopAttack() – Stops the attack animation.

void Toggle3rdPersonMode(bool flag) – Switches to 3rd person mode of attacking.

4.9.2. IRangeWeapon

An interface implemented by range weapons. Inherits from *IWeapon*.

Important Variables:

Transform projectileSpawn – The point at which the weapon's projectiles are supposed to be instantiated.

GameObject projectile – A prefab of the projectile that this weapon is supposed to shoot.

Transform targetTransform – The *Transform* of the entity at which the weapon is supposed to be shooting.

Vector3 target – The point at which the weapon is supposed to be aiming.

Important Functions:

void CastProjectile() – Shoots a projectile at the target.

4.9.3. IMidRangeWeapon

An interface implemented by mid-range weapons. Inherits from *IRangeWeapon*. Adds two overloads of the *Switch* function, which is used to switch between the weapon's melee and range form of attack.

4.9.4. WeaponEnd

This script is enabled when the unit is in 3rd person mode. It enables the weapon to deal damage by hitting another GameObject. It is used by every weapon in this game.

Important Variables:

List<GameObjects> cutObjects – A list of objects that the weapon has already intersected. This list is used to ensure that the weapon doesn't deal damage to a GameObject twice.

Important Functions:

void OnTriggerEnter(Collider other) – If the script is enabled and *other* is a non-trigger Collider which belongs to an enemy GameObject, the enemy receives damage.

4.9.5. Arrow

Important Variables:

Vector3 direction – The direction in which the arrow is supposed to be fired.

Important Functions:

void Start() – Uses the *AddForce* function, which is a function of the Rigidbody component. It takes a Vector3 argument and applies that vector as a force to the Rigidbody.

4.9.6. ArrowTip

A script attached to the tip of an arrow. It is used to stop the arrow when it hits something.

Important Variables:

GameObject exception – Stores the GameObject of the unit that shot the arrow, since the Arrow isn't supposed to be affected by that unit's Colliders.

Important Functions:

void OnTriggerEnter(Collider col) – Stops the arrow if *other* is a non-trigger Collider which doesn't belong to *exception*.

4.9.7. MagicBall

Important Variables:

GameObject boomEffect – A prefab of the effect that is to be instantiated when the ball hits something.

GameObject exception – Stores the GameObject of the unit that shot the ball, since it isn't supposed to be affected by that unit's Colliders.

Important Functions:

void OnTriggerEnter(Collider col) - Destroys the ball if *other* is a non-trigger Collider which doesn't belong to *exception*. Also instantiates *boomEffect* at the point of collision.

4.9.8 BasicSword

Important Functions:

void ClearCutObjects() – Clears the *cutObjects* variable of the *WeaponEnd* script attached to the tip of the sword. Is called from the sword's attack animation.

4.10. Spells

4.10.1. SpellTimer

This script is used to stop a spell after a given time interval.

Important Functions:

void StopSpell() – Stops the spell and destroys the SpellTimer GameObject.

void StopAfterTime(ISpellIcon icon, int time) – After the given amount of time, it invokes the *StopSpell* method.

4.10.2. DamagingAuraSpell

A spell which damages nearby enemy units.

Important Variables:

List<BasicUnitFunctions> enemies – A list of the *BasicUnitFunctions* components of enemies that have entered the spell's range.

Important Functions:

void Drain() – Drains the casters mana.

void DealDamage() – Deals damage to units in *enemies*.

void Awake() – Invokes the *Drain* and *DealDamage* functions repeatingly. *Drain* repeats every 0.5 seconds and starts after 0.5 seconds, while *DealDamage* repeats every 3 seconds and start immediately.

void OnTriggerEnter(Collider other) – If *other* belongs to an enemy unit, its *BasicUnitFunctions* are added to *enemies*.

void OnTriggerExit(Collider other) – *other* is removed from *enemies*.

4.10.3. TeleportEffect

This script teleports the unit when the effect to which it is attached is destroyed. This is done through its *OnDestroy* method.

4.10.4. HealSpellProxy

Important Functions:

void Update() – This function ensures two things. First, if the player left-clicks on a friendly unit, the Heal spell is cast on that unit. Second, if the player right-clicks somewhere or presses the tab key (thus exiting 3rd person mode), the spell gets deactivated.

void OnGUI() – Draws the heal icon next to the mouse cursor.

4.10.5. ShieldSpell

A script that ensures that the shield changes its alpha value, drains the casters mana and gets destroyed once the caster's mana runs out.

4.11. Other scripts

4.11.1. BodyPart

An enum of the body parts of a fighter unit. It is used by the icon scripts of the three weapons to determine what body part the weapon should be attached to.

4.11.2. ItemDatabase

This script is used by Loot Boxes (3.7.1.). It stores the icons of items that the Loot Box contains.

Important Variables:

List<IIcon> slots – A list of icons stored in the database. It is initially filled with 16 nulls (they represent empty slots).

4.11.3. LootBox

This script is used to spin the body of a Loot Box and to destroy it at the appropriate time. It is attached to the body of a Loot Box, not its root element.

Important Variables:

bool readyToGo – Indicates that the Loot Box has existed for 30 seconds.

Important Functions:

void Update() – Keeps spinning the body of the Loot Box. Also destroys the Box if *readyToGo* is true and no unit is currently viewing its contents.

5. Conclusion

5.1. Fulfilment of goals

In the introduction, I have stated seven goals that I wanted my game to fulfil. Here, I go over how well I managed to do that.

5.1.1. Create a basic RTS (goal #1)

My game fulfils the requirements of a playable RTS game. It allows the player to create buildings and units that he/she can use to beat the enemy. One common feature which is missing, and which would have made the 'strategy' part of the game more essential, is a more complex way of resource gathering. Although this was never a necessity, it does take away from the overall enjoyment value of the game.

5.1.2. One human player and at least one A.I. (goal #2)

There is one of each and more could easily be added with the *CreateAgent* function implemented in the *GameMaster* script.

5.1.3. 3rd person mode & combat system (goals #3 & #4)

These are the points at which I failed the most. I had originally intended for the 3rd person mode to be an essential part of the gameplay. As I was implementing the game though, I realised that for that to be possible, my game would have to be far more complex. This is because the main advantage of controlling a single character from up-close is that it can offer the player far more detailed manipulation of the character and of the environment. For that to be useful though, the character or environment have to be sufficiently complex.

The result is that all the fighter units do offer the option of playing for them in 3rd person mode, but it doesn't have much usefulness or entertainment value.

5.1.4. Spell & item systems (goals #5 & #6)

I have successfully implemented systems for storing and interacting with spells and items as well as a few instances of spells and items.

5.1.5. Expandability (goal #7)

My implementation offers straightforward ways of adding new kinds of buildings, weapons, items, units and spells. Creating new scripts for new entities should also be straightforward, given the way I separated the classes.

5.2. Possible Improvements

5.2.1. Combat system

As I have stated in the previous section, the combat system of my game could be greatly improved upon. Its strategic mode version works well, although there are some details that could be tweaked. For example, units always follow their targets unless either the target gets too far or the player commands the unit to stop. However, the more crucial part is the 3rd person version.

This version could be improved in many ways. For example, letters could appear somewhere on the screen which would indicate what button the player has to press to avoid an attack. Or there could be different kinds of attacks and each one could be countered by a specific defence move. Or there could be attack combos, or a myriad of other things. The main point would be requiring some sort of hand-eye coordination, as is common in action games.

The usage of range weapons could also be improved by providing a better way of aiming.

5.2.2. A.I.

The artificial intelligence in my game is very simplistic and can be easily defeated once the player knows how it works. One of the ways it could be made better would be by providing it with a broader range of reactions to different states of the game. This could be done via a finite-state machine. Each A.I. entity would have a set of states and conditions under which it would switch from one state to another. This is the technique which is usually used in RTS games.

Another approach that could be used here is called goal oriented action planning [7]. In this approach, an agent has a set of states and a set of actions. When given a goal, the agent searches the actions and picks a path that leads it to the goal. This approach is mostly used to control non-player characters in games, so it could probably be used to control the units in this game and maybe somehow combined with a finite-state machine which would control the strategic decisions and issue goals to the units.

5.2.3. Aesthetic changes

There are many aesthetic aspects of this game that could be improved. For example, the health- and manabars are always turning to look at the camera, which causes them to not be parallel to the top and bottom of the screen. Another example would be that when buildings are destroyed, they simply disappear, without some animation like units have. Yet another would be the fact that projectiles shot from range and mid-range weapons don’t necessarily hit their targets.

5.2.4. Expansion

Because I didn't aim to make a small but well-developed game, but rather chose to create only the basis of a bigger game, I only implemented the bare minimum of things necessary for this game to be playable. Therefore, a possible improvement would simply be – add more. Especially more spells and items, since they could greatly diversify the game and also make the 3rd person mode more interesting.

5.2.5. Developing the environment

The only function of the environment in my implementation is that it provides something for the units to move on and for the buildings to stand on. This means there is a lot of room for improvement. There are three main things that could be added – different types of environment, random objects that could be manipulated, and resources. The first two of these additions would provide the agents with more tools that could be used to defeat the enemy. They could also be used to make the 3rd person mode combat system more entertaining. As far as the resources go, there could be different types, like in Warcraft III, different combinations of which would be necessary to make different things.

5.2.6. Different maps

An important factor when considering the entertainment value of a game is how long it can take a person to discover all of it. Which is why it is good to have different maps – it makes the gameplay less repetitive. Every map adds some sort of novelty to the experience. It would therefore be good to add some maps to my game, since it only has one.

5.2.7. Levelling up

As it is, my system of levelling up just causes the unit's stats to increase. A better version could offer the player abilities of the unit that he/she could choose to unlock or stats that he/she could choose to improve.

5.2.8. Settings

The settings that I set in this game, like the time it takes to build individual buildings, or the amount of health each unit has, could be tweaked in a million different ways. I designed mine to make the game playable, but I am sure that a much better result could be achieved. For example, if I made the creation time of buildings and units long, that would give the player time and incentive to level up some of his/her units while he/she waits, which could make the gameplay more interesting. But maybe it would just make it boring because player's would have to wait around for long stretches of time for their buildings to be completed.

Attachments

Attachment A: References

[1] jMonkeyEngine website (18.7.2019)
http://jmonkeyengine.org/

[2] Unity website (18.7.2019)
https://unity.com/

[3] Unreal Engine website (18.7.2019)
https://www.unrealengine.com/en-US/

[4] Unity asset store (18.7.2019)
https://assetstore.unity.com/

[5] Unity user manual (18.7.2019)
https://docs.unity3d.com/Manual/UnityManual.html

[6] Unity life cycle (18.7.2019)
https://docs.unity3d.com/Manual/ExecutionOrder.html

[7] Goal oriented action planning (18.7.2019)
https://gamedevelopment.tutsplus.com/tutorials/goal-oriented-action-planning-for-a-smarter-ai--cms-20793

[8] Unity archive (18.7.2019)
https://unity3d.com/get-unity/download/archive

[9] Unity system requirements (18.7.2019)
https://unity3d.com/unity/system-requirements

Attachment B: Archive description

The CubeFighter.zip file contains a folder called Scripts and a file called CubeFighter.unitypackage.

The Scripts folder contains all the scripts that I programmed for this game. They are split into the same categories that I used in chapter 4.

The CubeFighter file contains my entire Unity project. I made this project on Windows 10 in Unity version 2018.2.11f1, so trying to open it in any other environment may cause problems. A different operating system shouldn't be much of an issue, but a different version of Unity could cause some deficiencies. The version of Unity I used can be downloaded from the Unity archive [8]. The version of Visual studio I used was Visual studio 2017 15.7.5, but using a different version should cause no problem, since Unity has its own compiler.

In order to open this project on another device, first create a new Unity project. Then import the file through Assets > Import package > Custom package.

Once the project is opened, the assets folder is shown at the bottom of the screen. It contains these folders and files:

Animations – Contains animators and animations.
Materials and Textures
Resources – Contains prefabs and icon images.
Scenes – A folder generated by Unity. Contains the games scene data.
Scripts
B13a – The song used in this game.
MinimapTexture – The texture onto which the image that the MapCamera sees is projected.
New Terrain – Data about the game's terrain.

Attachment C: User documentation

Requirements

The game was developed for and tested on Windows 10. However, it should work on some macOS and Linux systems too. The specific system requirements for running Unity games such as this one can be found at the official Unity system requirements page [9].

Installation

This game requires no installation.

Starting the game

The Fighter.exe file starts up the application. Upon starting, a window is displayed which allows the user to choose graphics settings like resolution and quality. The user can start up the game by clicking the 'Play!' button in the lower right corner of the window. The window also allows changing some input settings, but these don't actually change any settings of the game, so they can be ignored.

Menus

All the UI elements in the menus can be interacted with by using the left mouse button to click or drag them.

Main menu: Appears automatically when the game is launched. Contains 3 buttons – the 'play' button which starts the game, the 'settings' button which transfers the user to the Settings menu, and the 'quit' button which quits the application.

Settings menu: Can only be accessed through other menus. Contains a slider and a 'return' button. The slider controls the volume of the music being played (it is set to zero by default). The 'return' button returns the user to the previously displayed menu.

Pause menu: Appears if the user presses the escape key during the game. Contains 3 buttons – the 'continue' button which resumes the game and the 'settings' and 'quit' buttons which fulfil the same function as in the Main menu. The user can leave the menu by pressing the escape key again.

Game Over menu: Appears when only one team remains. Contains 3 buttons – the 'play again' button which restarts the game and the 'settings' and 'quit' buttons which fulfil the same function as in the Main and Pause menus.

Game environment

GUI

Minimap: The minimap is a square situated in the lower left corner of the screen. It provides the player with a view of the entire game world. The player's units and buildings are displayed on the map as blue dots, while the enemy's units and buildings are red dots. The Monster units, which don't belong to either team, are violet dots.

Menus: In the lower right corner of the screen is the main game menu (MGM for short). This menu consists of 9 squares which can display different icons that the player can interact with. Besides MGM, there are 3 other menus that appear in the game under various conditions. They are: the spellbar, the equipment menu and the item database. They all consist of slots like the ones in MGM which can be filled with icons. The spellbar appears at the bottom of the screen while the equipment menu and the item database appear in the centre.

Sliders: In the upper right corner of the screen, a unit's level slider can appear which displays the unit's current level and the amount of experience that it has gained since last levelling up. Above each unit are its health- and manabars which show how much health and mana the unit has left. The healthbars are also present above buildings.

Text fields: In the upper left corner of the screen, the amount of the player's resources is displayed in white text. If the player tries to do some action which requires more resources than he/she currently has, a notification appears in red text in the middle of the screen.

Entities

There are two basic types of entities in this game which form the backbone of the gameplay – units and buildings.

Units:

- Worker: The only unit incapable of fighting. It allows the player to build all the buildings available in this game. Can be created in a Base building.
- Warrior: A melee fighter unit. Can be created in a Barracks building.
- Archer: A range fighter unit. Can be created in a Barracks building.
- Mage: A mid-range fighter unit. Can be created in a Barracks building.

- Monster: A melee fighter unit. The only unit that cannot be created in any building. Units of this type don't belong to any team.

Buildings:

- Base: The sturdiest building. Allows the player to create Workers.
- Barracks: Allows the creation of fighter units.
- Farm: Produces resources. The most vulnerable building.
- Tower: An offensive structure. Can attack enemy units/buildings by shooting arrows at them.

Rules

The goal of this game is to destroy all of the enemy's units and buildings. In the beginning, every agent starts out with a single Base and 1000 resource units. These units are a sort of currency that can be exchanged for the creation of units and buildings. In order to win, the player must make use of the units and buildings that he/she creates to destroy the enemy.

All fighter units can level up by killing other units which increases their stats, like power, speed, or amount of health points. This means, that besides just using resources to generate more units and overpower the enemy with numbers, the player can fight quantity with quality.

Game controls

The game consists of two modes – the strategic mode and the 3rd person mode. Each offers different ways of interacting with the world and each has different controls.

Strategic mode

In this mode, the mouse is the player's most important means of communication with the game.

Selecting entities: By left-clicking a unit or a building, the player can select it. By left-clicking and dragging the mouse cursor, the player can create a rectangle and select all the units inside it. This works only for units though, not buildings. Left-clicking causes the selected entities to be deselected. A selected unit has a green highlight around it. A selected building has an orange arrow hovering above it.

Menu: If the player has selected a single entity, be it a unit or a building, the entity's icons are shown in the main game menu. The player can interact with these icons by left- and sometimes also right-clicking them, although these two actions are never interchangeable. It could be loosely said that left-clicking an icon means activating

it, while right-clicking means deactivating it. Pointing at an icon with the mouse cursor causes its description to appear.

If the selected entity is a building, one of the displayed icons is the 'destroy building' icon, which allows the player to destroy that building by left-clicking the icon. The Base and the Barracks also display other icons, which allow the creation of units. Left-clicking these icons causes a unit to start being created, while right-clicking them causes a unit to stop being created.

If the selected entity is a Worker, the displayed icons are the buildings that it is capable of building. Left-clicking these icons causes an outline of the building to appear. The player can then either command the Worker to go build that building by left-clicking somewhere on the map where the building can be built (indicated by the fact that the outline remains green), or just make the outline disappear by right-clicking anywhere. If the outline turns red, it means that the corresponding building cannot be built there. If the player commands the Worker to go build a building, but then right-clicks somewhere while still having the Worker selected, the Worker ignores the command to build.

Finally, if the selected entity is a fighter unit, the icons displayed in the main game menu are the items in its inventory. They can be used/equipped by left-clicking them and destroyed by right-clicking.

Navigating units: Once the player has one or more units selected, he/she can navigate them across the map by right-clicking on the place he/she wants them to go. Even if the player sends some units to some destination, they can still end up engaging in a battle with some enemy, if they get too close to it. However, if the player sends them to some destination while they are fighting, they go there without engaging in further battle.

If the player right-clicks on an enemy building/unit while having fighter units selected, they go attack that entity. However, only if the entity is a building do they not engage in battle with other enemy entities that they might meet along the way.

Camera movement: The player can move the camera in 4 ways. First, by placing the mouse cursor near any of the edges of the screen causes the camera to move in that direction. The camera cannot be moved outside of the map though. Second, the player can transport the camera to any point on the map by clicking on the desired location on the minimap. Third, by holding down the R key and then pressing any of the arrow keys, the player can rotate the camera. Finally, by scrolling the mouse wheel, the player can make the camera zoom in/out. The camera cannot zoom in/out arbitrarily though. If zooming out any further would cause it to get outside of the bounds of the map, it stops, and there is also a limit to how close it can get by zooming in.

Base and Barracks: When the player has either a Base or a Barracks selected and right-clicks somewhere on the map, the building's 'spawn destination marker' is moved there. This element tells the units where they should go after being instantiated.

In this mode, the player can switch to a unit's 3rd person mode by selecting only that unit and pressing the tab key.

3rd person mode

In this mode, the keyboard becomes much more significant.

Movement and attack: The unit which the player is controlling can be moved by pressing the W, A, S and D keys (W – go forward, A – turn left, S – go back, D – turn right).

The player can trigger the unit's attack by clicking the left mouse button. If the unit has a range or a mid-range weapon equipped, the weapon shoots a projectile at the point on which the player has clicked, unless he/she hasn't clicked on any GameObject. In that case the weapon fires in the last direction in which it has fired.

If the given unit has a range weapon equipped, pressing the Q key causes it to switch to the aim camera. This camera can be moved simply by moving the mouse. The player can still move the unit with the W, A, S and D keys while using this camera and fire projectiles using the left mouse button.

Menus: The main game menu permanently shows items in the unit's inventory. They can be interacted with the same way as in the strategic mode – left-clicking them uses/equips them, right-clicking them destroys them.

Pressing E causes the equipment menu to appear which shows what items are equipped by what parts of the unit's body. In this case, left-clicking the item icons does nothing while right-clicking them unequips them and places them inside the unit's inventory.

In this mode, the player can also interact with the unit's spellbar, which is displayed at the bottom of the screen. It displays icons corresponding to different spells, which can be cast by left-clicking the icons and, in some cases, deactivated by right-clicking them.

Camera: Pressing any of the arrow keys causes the camera to rotate. However, unlike in the strategic mode, the camera rotates around the unit, not around itself. The camera can also be zoomed in/out by scrolling the mouse wheel.

Loot Boxes: In 3rd person mode, the player can inspect the insides of loot boxes. This can be done by getting close enough to the loot box and left-clicking it. A 4 by 4 matrix of squares then appears in the middle of the screen, some of which have item icons in them. The player can store these items inside the unit's inventory by left-clicking them.

In this mode, the player can switch back to strategic mode by pressing tab.