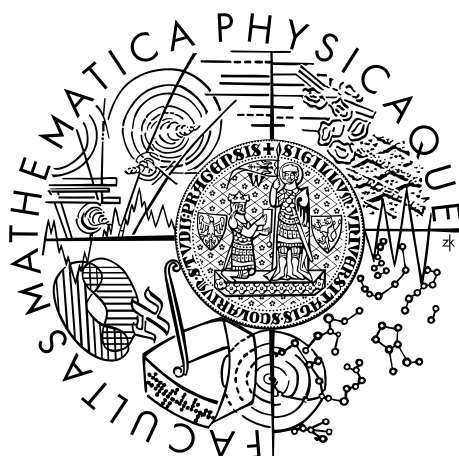Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS

Jaroslav Jindrák

The Dungeon Throne: A 3D Dungeon Management Game

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Pavel Ježek, Ph.D

Study programme: Computer Science

Study branch: Programming and Software Systems

Prague 2016

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 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....   signature of the author

Title: The Dungeon Throne: A 3D Dungeon Management Game

Author: Jaroslav Jindrák

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D, Department of Distributed and Dependable Systems

Abstract: Abstract. TODO:

Dedication. TODO:

Contents

1. Introduction

Until the release of Dungeon Keeper[1] most well known fantasy video games have allowed the player to play as various heroic characters, raiding dungeons filled with evil forces in order to aquire treasures and fame. In Dungeon Keeper, however, we join the opposite faction and try to defend our own dungeon (along with all the treasures hidden in it) from endless hordes of heroes trying to pillage our domain. Although we can still play the original Dungeon Keeper today, we cannot change its data or game mechanics in any easy way so this thesis aims to recreate and modify the original game and extend it to have an easy to use programming interface that will allow such modifications.

1.1   Dungeon Managment Genre

Dungeon Keeper was the first game released in the dungeon management (DM) genre and since our game is going to be based on Dungeon Keeper, we should design it with the elements of its genre in mind. Since the definition of this genre has to our knowledge never been formally documented by the creators of Dungeon Keeper, we are going to create a list of basic elements of the genre based on the gameplay of the original game.

In Dungeon Keeper, the player's main goal is to build and protect his own base, called the dungeon. They do so by commanding their underlings (often called minions), whom they can command to mine gold, which is a resource they use in order to build new rooms and cast spells (in the game's sequel[2], mana was added into the game as a secondary resource used for spell casting), which could be researched as the game progressed. They would then use creatures spawned in their buildings as well as their own magic powers to fight intruders in order to protect their dungeon. From this brief gameplay summary, we can create list of the most basic design elements, which can be found in dungeon management games:

(E1) Resource management

(E2) Dungeon building

(E3) Minion commanding

(E4) Combat

(E5) Player participation in combat

(E6) Research

Now that we have created a list of the genre's basic design elements, let us have a look at how they were implemented in games from the Dungeon Keeper series.

---

[1]Bullfrog Productions, 1997

[2]Dungeon Keeper 2, Bullfrog Productions, 1999

Resource management

In the original Dungeon Keeper, the player used gold as their primary resource. They would have it mined by their minions and use it to build new rooms and cast spells. While having a single resource for everything may bring simplicity to the game, it also means that once the player runs out of gold, they will not be able to directly participate in combat due to their inability to cast spells. It is for this reason that we are going to use the resource model of Dungeon Keeper 2 and have a separate resource – mana – that will be used to cast spells.

Dungeon building

The term dungeon building in Dungeon Keeper refers to tearing down walls in order to create new rooms which the player's minions then claim and the player can place new buildings in. These buildings can then act as gold storage, spawn new creatures or be used as traps that negatively affect attacking enemies. Since this is a central theme in Dungeon Keeper (and dungeon management games in general), we should try to implement our building model to resemble this one.

Minion commanding

The term minion commanding stands for giving your minion tasks such as to move somewhere, attack an enemy or tear down a wall. In Dungeon Keeper, most of these commands were implemented as spells – although mining was done by simply selecting blocks, which could unfortunately cause minions to destroy walls the player accidentally selected. To providea unified interface to minion tasks and to avoid the accidental block destruction, all commands the player can give should be in our game implemented as spells.

Combat

In Dungeon Keeper, the player's dungeon is under frequent attacks by enemy heroes. These enemies attack the dungeon in groups with a delay between each two attacks, in a similar way to tower defense games (e.g. Orcs must die! [1] and Dungeon Defenders [2]), where they are often referred to as *waves*. Each wave of enemies can consist of different types of heroes and the delay between waves can differ, too. Since the players of the original Dungeon Keeper are already familiar with this wave system and it may also satisfy the players of tower defense games, we will be implementing it in our game.

Player participation in combat

During the fights with enemy heroes, the player in Dungeon Keeper can cast various spells to affect the outcome of the battle. These spells can have various forms from spawning creatures, damaging enemies and healing minions to destroying walls and throwing meteors. If we want to create a similar spell system, we need to support these different types of spells, e.g. targeted spells that affect a single entity, positional spells that can summon a creature or global spells, that simply have an effect on the game once they are cast.

Research

In the original Dungeon Keeper, researching new spells and buildings was done through a special type of room, called the library, which can be seen in Figure 1.1. There, various minions could study and achieve new advancements for the player after a certain amount of research points – which was different for different spells and buildings – was gathered. But this design decision brought a negative element into the game, because once the library was destroyed by the heroes, the player was unable to perform additional research and could not even use the ability to cast some spells. This is why we are going to use a different approach, called the research tree (or sometimes the technology tree), first used in the turn-based stragety game called Sid Meyer's Civilization[3], as pointed out by Tuur Ghys in his article about technology trees [3]. This approach separates the research from the events happening in the game and allows the player to research new advancements for a resource (e.g. gold), with each new technology, building, creature or spell being able to have prerequisites.

Figure 1.1: Warlocks researching in the library.
Source: http://dungeonkeeper.wikia.com

1.2   Modifiability in Games

One of our basic goals is for our game to be modifiable, which means to provide tools – often called *modding tools* – to our players that will allow them to create modifications – often called *mods* – that other players can install and which can change or add elements to the game.

This can increase the replay value of the game as after finishing it, more missions, characters, game mechanics, abilities, items or even game modes can be easily downloaded and installed from internet. Since we want our game to be modifiable as much as possible, an important topic we need to decide on is

---

[3]Developed by MPS Labs, 1991

which parts of the game we will allow the players to modify. An example of an easily modifiable game is Minecraft [4], a 3D sandbox game in which the player has to survive in a procedurally generated world. Let us now examine the basic concepts of the game before we investigate concrete examples of mods and how they changed the game in the following sub chapters. As a reference, a screenshot of Minecraft can be seen in Figure 1.2.

Figure 1.2: Screenshot of Minecraft.
Source: http://www.neoseeker.com

In the screenshot we can see a player standing in the game's world, equipped with an iron sword that can be used for self-defense when the player is attacked by enemies or to kill animals in order to aquire food required to restore one's health, standing next to a hill and a couple of trees. Everything in the game is made of *blocks*, which are items that can be placed in the world and – with some exceptions[4] – destroyed so that the player can place them in their inventory. They can either be used for building or they can be interactive, which the player can right click on to trigger their functionality – e.g. show a special window or change their state. An example of an interactive block is the chest block, which provides additional item storage when the player interacts with it.

In the bottom part of the screen we can see the hotbar, which the player can use to store commonly used items like tools – e.g. a pickaxe, which is used to mine blocks, or an axe, which is used to cut down trees and allows the player to harvest wood – and blocks, which can be used for building or crafting.

Crafting is an important part of Minecraft. It allows the player to create tools and complex blocks by combining one or more crafting ingredients – e.g. wood, stone, iron or food. The crafting system in the game is based on different crafting recipes, which restrict what items and in what composition on the game's crafting window – either a two by two window in the player's inventory screen or three by three window shown when the player uses a special interactive block called the crafting bench.

Figure 1.3 shows an example of such crafting recipe, specifically the pickaxe crafting recipe. The existence of this recipe in the crafting system means that if a player interacts with the crafting bench and places two wooden sticks and three

---

[4]E.g. the bedrock block, which covers the bottom layer of the world and prevents the player from falling outside of the world's boundaries
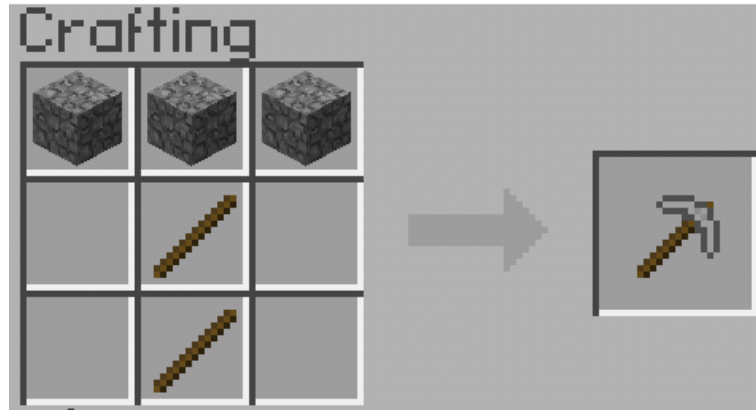
Figure 1.3: An example of a crafting recipe in Minecraft, the player can create a stone pickaxe by combining two wooden sticks with three stone blocks. Source: http://thecoolestminecrafters.weebly.com

stone blocks – although different resources like iron or gold bars can be used for a different type of pickaxe – in the layout shown in the picture, the crafting bench will produce a stone pickaxe in exchange for the resources.

1.2.1 Mod examples

There are thousands of different minecraft mods – for example, the mod repository at Curse.com [5] contains over 4 thousand mods and the one at Planet-Minecraft.com [6] contains even over 7 thousand different mods. To find the right moddable features of our game, we will now look at some examples of these mods and see how they change Minecraft.

Industrial Craft 2

The first example is called Industrial Craft 2 [7]. This mod adds new blocks and tools as well as a new game mechanic – the concept of electricity – to the game. The items added by the mod range from simple ones like new types of armor or new tools to complex blocks like various electricity generators and automatized machines.

An example of such machine is the electric furnace. In the game without mods, the player gets iron and other metals in the form of a raw ore when they mine a block containing the metal. To use the metal in a crafting recipe, the players needs to smelt the ore into a bar using the furnace, which can be powered by wood or coal. The electric furnace uses electricity for power and can be powered by an electricity generator – e.g. a solar panel – which removes the necessity to repeatedly place wood or coal in the machine.

In Figure 1.4, we can see a simple setup in which a couple of solar panels are connected to an electric furnace and other machines via glass fibre cabels. We can also see another interactive block added by the mod – the battery block. The battery block stores electricity up to a certain capacity and continues powering the machines even when its electricity input stops, which in the case of solar panels can happen during the night.
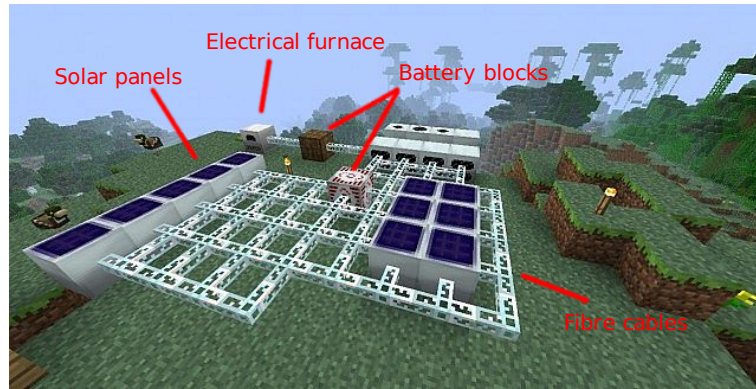
Figure 1.4: A simple setup powering multiple machines including an electrical furnace using solar panels.
Source: http://www.planetminecraft.com

Using these machines, the game can be changed from its rather primitive setting – in which the player mines resources with simple tools and builds simple houses – to a modern setting with automated mining and resource processing machines. If we want to allow the players of our game to create similar modifications, they should be able to change existing and create new buildings – our equivalent of Minecraft's blocks – which includes changing their appearance, the way they function – e.g. what type of minions they spawn – and how they interact with enemies.

Computer Craft

Another mod, called ComputerCraft [8], added fully functional computers into the game. These computers can be used to create password protected doors, write programs and games in the Lua programming language [9] or even connect to internet services such as IRC[5] or telnet[6].
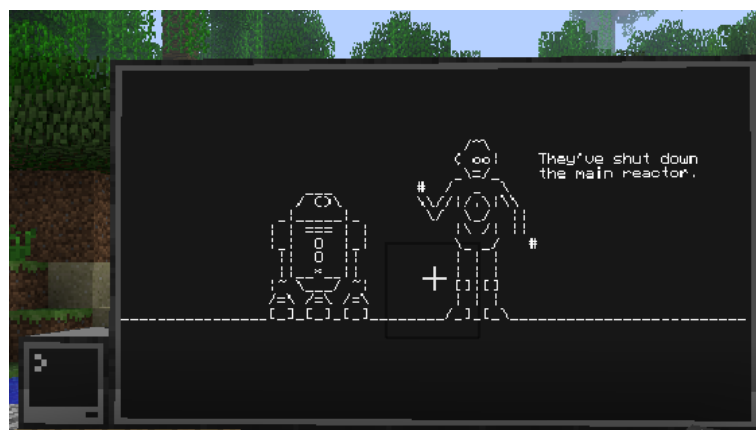
Figure 1.5: A computer in Minecraft playing the ASCII version of the movie Star Wars: A New Hope.
Source: http://www.minecraft-modding.de

---

[5]Internet Relay Chat, an open communication protocol.
[6]A client-server protocol, that allows bidirectional text-oriented communication through a terminal.

An example of such computer in minecraft can be seen in Figure 1.5. There, a player has connected a computer to a screen and used the computer to connect to a server via telnet. The server then sent back messages that, when displayed sequentially, played the first Star Wars movie.

Figure 1.6: A turtle equipped with a diamond sword about to attack a pig. Source: http://www.computercraft.info

This mod also added robots into the game. These robots – called *turtles*, inspired by the Logo programming language – could be programmed using Lua scripts to automate various tasks. The task a turtle fulfills is determined by the tools it is equipped it. If, for example, the player gives it a pickaxe, the turtle will then be able to mine blocks and being equipped with an axe will allow it to chop down trees. In Figure 1.6, we can see another type of turtle, this time equipped with a sword. The mod provides the player with an API in which they can write instructions for the turtle. If, for example, these instructions were in this case 'walk three times forward and attack' – written in Lua – the turtle would walk forward and killed the pig standing in front of it.

We have already decided to allow our players to modify the buildings in our game, based on the machines in Industrial Craft 2. From Computer Craft, we are going to take the concept of minion and enemy modification. This means that the players of our game should be able to change the attributes of both minions and entities in the game, including attributes such as health and damage, abilities, spells and most importantly, their overall behavior (i.e. their artificial intelligence).

Team Fortress 2 Map

Aside from adding items and creatures, entire new games can be created within a modifiable game. One of the interactive blocks in Minecraft is command block, which can execute commands in the game's developer console – e.g. teleport the player to a certain position or give him items – when the player right clicks it.

One of the players of the game, called Seth Bling, used this block to create a map which alters the way the game is played [10]. For example, even though the unmodded game has no concept of character classes, he used the command block to equip the player that clicked on it with a special set of tools and weapons, using the command to add items. With multiple of these blocks – each with a different set of items – allowed him to emulate classes in the game. Then, using the same block with different setting, he created control points which the players

could stand on to capture them. Using these control points he was then able to alter the game's winning condition, which was originally to kill a specific enemy, to a requirement team of players having all control points in the world under their control. Once all control blocks were controlled by the same team, another command – requesting the game's end – would be executed.

The result of these modifications was the change of Minecraft to look and play like another game, called Team Fortress 2 [11]. Team Fortress 2 is a 3D first person shooter, in which two player teams fight against each other in various game modes, e.g. the aforementioned point control. Each player in the game can choose from a wide selection of character classes, each with its own attributes and weapons.

Unlike Industrial Craft 2 and Computer Craft, which only added new items and non-player characters into the game, this map mainly altered two aspects of the game. Firstly, it changed the way the player plays the game by limiting them to a specific number of different classes they can play, bringing diversity to the game's online play. Secondly, it changed the way the game ends, i.e. under what condition the player wins or loses.

From this map, we are going to take the ability to modify these exact two aspects. While the change of the game's end condition can be done similarly to the way it was done in this game – i.e. executing a custom command to win or lose the game, the change of the play style can not be done by item restriction because in our game the player has no items. The way that, in our game, the player interacts with the world is through their spells – which include entity commanding. This means that alongside building, minion, enemy and end condition modification, our players should be able to change their spells and create new ones. These changes should be able to affect the gameplay, so they should range from simple damage and range modification to complete changes of the effects these spells have on the world.

One important note is that this change of gameplay was not done by creating a mod but was created as a map for the game, which the players could download from internet and then simply load in the game without any installation. This way of distribution of changes to the game, alongside modding, creates another way for the players of a game to create and share modifications of the game. Since we want our game to be modifiable, we should implement a way for our players to create similarly game changing maps in our game, i.e. we should store our levels in a format that will allow later modifications, including actual changes to the game's elements.

Conclusion

In conclusion, we can see that the ability to modify a game can help said game to grow even when its development has stopped or is focused in different areas (e.g. security, stability). Since we want to give this ability to the players of our game, our modding tools should allow them to change some of its data, including but not limited to:

- Minions and enemies – e.g. changing attributes such as health and damage, displayed models, behavior and spells they cast.

- Buildings – e.g. changing their size, models, types of creatures that they spawn and, in the case of traps, their interaction with enemies.

- Spells – fully changing the effect of a spell, e.g. from simple damage dealing to spawning a meteor shower.

- Goals of the game – changing requirements for winning the game or the reasons for a loss.

Besides changing data of entities – e.g. creatures, buildings or spells – the players should also be able to create new types of these entities.

The game on its own, like Minecraft, should also be fully featured, offering enough of these entities on its own so the players do not need mods to actually play the game. Additionally, since our game, like Dungeon Keeper, will have scripted waves of enemies attacking the player's dungeon, the also should allow our players to alter the wave composition – that is, which types of enemies compose the different groups attacking the player's dungeon – and delays between the waves. Last, but not least, we must not forget that players do not necessarily have to be (and often are not) programmers, so our game should provide an easy way to install these modifications.

1.3 Competetivness of the game

We have now investigated the main design elements of Dungeon Keeper and how we are going to implement them in our game. The last thing we have to realize is that since we are trying to satisfy the players of the original game, the resulting product of our work should be a full game and not just a prototype. This means that the game should offer full singleplayer experience, with relatively intelligent enemies and the ability to not only win the game, but to also lose. Also, for the game to be competetive with other titles, it should be performant, achieving at least the minimum acceptable framerate, defined as 25-30 frames per second by Shiratuddin, Kitchens and Fletcher [12], on both new and older hardware.

1.4 Thesis Goals

The main goal of this thesis is to design and implement a modifiable 3D dungeon management game using the design elements (**E1**) – (**E6**).

In addition to the main goal, the game should complete the following list of goals:

(**G1**) The game has to be a full competetive product, not a prototype.

    (**G1.1**) It has to be performant, achieving high framerate even on low end computers.

    (**G1.2**) It has to offer full single player experience, including enemies with scripted behavior and a chance to both win and lose.

    (**G1.3**) It has to contain a variety of entities, spells and buildings even without mods.

(G2) The game has to be highly modifiable, providing an easy to use modding interface for players.

    (G2.1) The mod creators must be able to create new entities (including their behavior), spells and buildings and to change characteristics of already predefined entities, spells and buildings including, but not limited to health, damage, model, behavior and abilities of an entity, effect of a spell and which kind of minion does a building spawn.

    (G2.2) They must also be able to alter the game progression by defining enemies that spawn and delays between them.

    (G2.3) The game should also support the creation of custom levels.

(G3) The mods for the game have to be easily installable even by players without any programming knowledge.

2. Problem Analysis

In the introduction section of this thesis, we defined a list of design elements **(E1)** – **(E6)** that we are going to implement in our game. We have also presented the notion of modifiability and examined some of the numerous Minecraft mods to see which parts of a game can be modified. In this section, we are going to look at the different tools, libraries and engine design possibilities that could potentially be used to implement our game.

2.1 Modding Tools

To satisfy our goals **(G2)** and **(G3)**, which require us to provide modding tools to our players that will allow them to create easilly installable mods, we now need to decide the form of these tools. They should allow the mod creators to create and modify entities to the extent required by our goal **(G2.1)** and to alter the game progression as required by our goal **(G2.2)**. Additionally, to satisfy our goal **(G3)**, the resulting mods should be easily installable.

Editor

A game editor is a tool that is commonly used to modify a game. It can be either an external application or it can be a part of the game. In most cases it's used to edit maps or scenarios for the game. But since the maps in our game are created by the player – by destroying walls and creating buildings – we are more interested in a different aspect of editors and that is changing entities in the game.
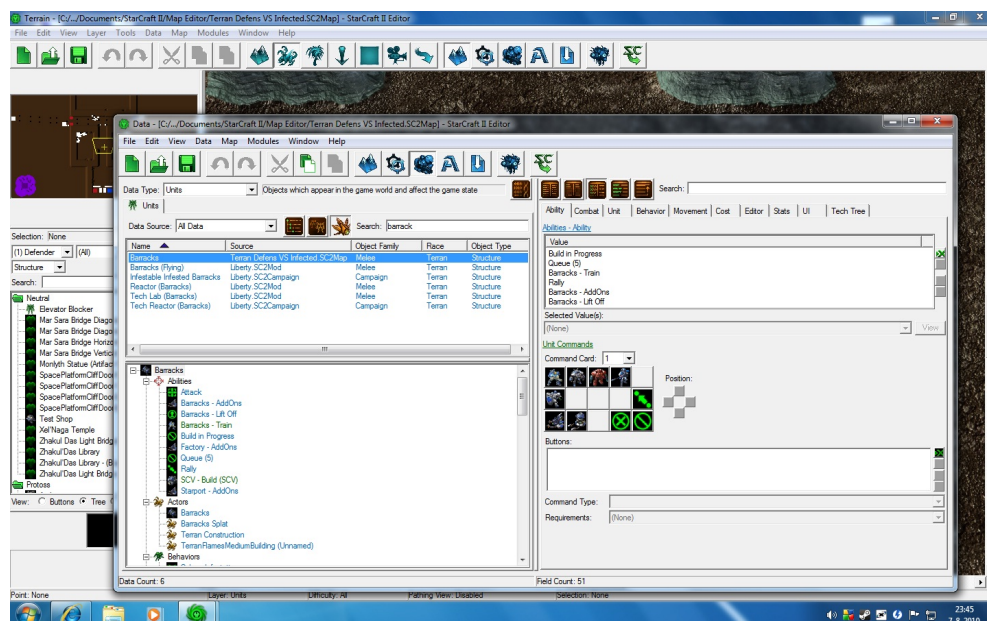

Figure 2.1: Besides map editing, some editors – like the Starcraft editor in this figure – can be used to edit entities.
Source: http://www.darvo.org

In Figure 2.1 we can see an example of such editor. It allows its user to select an entity and change its attributes, such as health, placing cost, type, model and the user can even select which predefined abilities and behavior the entity uses. An editor could easilly be used to define the composition of enemy waves and the delays between waves, satisfying our goal **(G2.2)** and would allow easy mod installation – as required by our goal **(G3)** – by producing files that can be placed in the game's directory.

While this approach could be easily implemented using configuration files for entities, our goal **(G2.1)** requires the ability to alter the behavior of entities which includes creating entirely new behavior and thus only selecting predefined AI won't suffice. To achieve this, we could require our users to write the behavior using a scripting language, which wouldn't be much different from the option that will be examined after this one. Alternatively, we could create a graphical tool that would allow the user construct the behavior out of blocks representing decisions and actions which would then be translated to source code – again, requiring a scripting language interfaced to the engine. An example of such graphical tool is the blueprint system used by Unreal Engine [13].
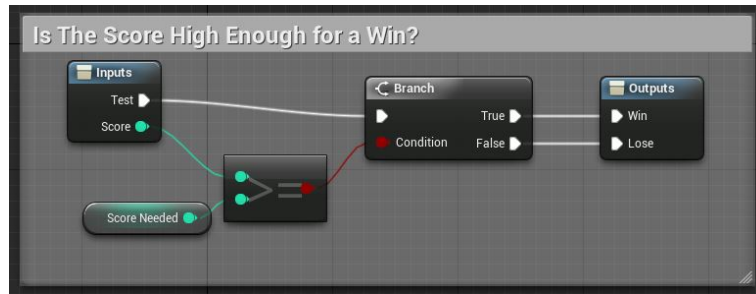

Figure 2.2: A simple blueprint checks if a person passes a test.
Source: http://www.unrealengine.com

In Figure 2.2 we can see an example blueprint created in Unreal Engine, it comprises interconnected blocks that represent conditions, loops, actions and other constructs that can be found in a typical programming language. The user uses these blocks to create programs in a more user friendly manner. Unreal Engine then uses these blueprints to generate C++ code that is then used in the game.

API

This option would require us to create an interface in which functionality of the engine binds to functions that can be called from a scripting language embedded within the engine. This would allow our players to write scripts that can be loaded at the start of our game or during its runtime and can change anything provided by the interface.

An example of such API can be found in the game Starbound [14], which allows its players to create modifications by using its Lua API to write scripts. An example of a simple Starbound script, which defines a new object in the game that spawns a chicken when the player interacts with it, can be seen in Listing 1. Here, the mod creator defined two functions called *init* and *onInteraction*. These two functions are defined by the game and are called by it – *init* is called when the

object is placed into the world and *onInteraction* is called whenever the player interacts with the object. The game also provides each mod with several tables [1] that help the mod to interact with the game world – in this case the mod uses the *object* table, which represents each instance of the spawner object, and *world*, which represents the game world and is shared between all objects.

```
function init()
    object.setInteractive(true)
end

function onInteraction()
    world.spawnMonster("chicken",
                       {0, 0}, {level = 10})
end
```

Listing 1: A simple script that represents an interactive monster spawner. When the player interacts with this object it spawns a level 10 chicken at the absolute coordinates (0, 0).

This example shows us a common interface used for mod making. The game provides a set of functions and data and the mod implements functions required by the protocol used for communication between the game and the mod. These functions are then called by the game when an event – which the called function is assigned to – occurs. Since our goals require our modding tools to be easy to use, we think that if we decide to create a modding API, it should use a similar approach, seeing as it might be familiar to mod creators because games like Starbound, Cities: Skylines [16], Factorio [17] and others use this approach.

This option allows us to provide a large amount of our engine's functionality and data wrapped in an easy to use interface, which can lead to the ability to easily modify it and thus satisfying our goal **(G2.1)**. This interface can also contain functions that handle the game's wave system and allows the mod creators to modify wave composition and delays between waves, which would satisfy our goal **(G2.2)**. To satisfy the last of the three goals, **(G3)**, we can design the scripting system of our game in a way that only requires the players to place a downloaded script in a directory and register the script in a config file.

Besides making the game modifiable, this approach can avoid long recompilation times because parts of the game that are not performance ciritcal can be written in an embedded interpreted language. This can lead to faster development [18], since recompilation is needed only after engine modification. This also allows fast testing and prototyping of new features and mods without the need to restart the game as many of these interpreted languages can execute pieces of code passed to them as strings.. Additionally, using an interpreted language would make the mod creation process easier for non-programmers [19] as they often have an easier to understand syntax.

---

[1]A table is the only data structure in Lua, it is a key value associative array that can be used to represent ordinary arrays, sets, queues and other data structures. [15]

2.1.1 Conclusion

While an editor would open our modding tools to a broader community than an API would, implementation of a graphical programming language similar to the blueprint system in Unreal Engine, which would be needed to satisfy our goal **(G2.1)** that requires the ability to define new behavior, would be a much larger task than to implement an interface for an already existing programming language. For this reason, we decided to choose the second option – creating an interface to an embedded language – as the modding tool we are going to provide to our players.

2.2 Programming Language

The first tool we need to decide on is the programming language we are going to write our game in. This choice affects multiple aspects of the final game. These aspects can include, but are not limited to:

- Performance: Interpreted languages tend to be slower than compiled languages, but this does not to be always true due to the existence of Just-In-Time – often abbreviated as *JIT* – compilers, which can provide compilation to machine language at program start and runtime optimizations to increase the performance.

- Speed of development: Lower level languages often require the implementation of tools that are provided by the standard libraries of higher level languages.

- Modifiability: Some programming languages – e.g. interpreted ones – provide means to alter the source code at runtime, while other require recompilation or the use of an embedded language.

The programming language that will be used to create our game needs to have one or more libraries that will allow us to create 3D graphics and be fast enough to offer at least the minimum acceptable framerate while rendering game objects to the screen, updating the state of the game and processing user input. It should also allow our players to modify the game even if they only have the distributed version of the game – meaning it should be able to load code it was not originally compiled with.

Since mod development requires testing of the mod's functionality in the game, it would be beneficial if the game allowed our mod creators to change the game's mechanics and data at runtime so that they do not need to restart the game to see what effect does a change in their mod have on the game. This means that the ability to execute a piece of code input as a string or load source files during runtime is a feature our programming language should provide. Lastly, the language should allow us to create a modding API that can be provided to our users as discussed in the previous section.

Aside from these important characteristics, the language should also be easy to use by those of our players that decide to modify the game.

2.2.1 Native Language: C++

C++, the first language we are going to look at and also the language we ended up choosing, was for a long time the industry standard when it comes to video games. One of the main reasons for this was that it is – unlike some of its rivals, e.g. Java – a language that is native, i.e. compiles directly into machine code of a specific processor, which generally results into faster executed code.

This benefit of the language, while still present, got weakened by the rise of JIT compilers. Java can now make use of just-in-time compilation and C# was even originally released with a JIT compiler. This means that both of these languages can compile their intermediate language the first time it's executed. Since by the time this compilation takes place the compiler knows what operating system, architecture and hardware specification it compiles for it can provide optimization that a C++ compiler cannot perform, achieving comparable execution speeds. Additionally, both language have seen an increase of game development related middleware, released commercially successful games written in them and addition of new interesting features that make them to see more viable in the eyes of game developers.

Even though the performance gap between these languages got reduced, the era of C++ being one of the most used programming languages in game development has resulted into an abundance of game programming related materials like libraries, tutorials and books.

One of our main requirements for a programming language is the ability to load code it was not originally compiled with. C++ allows this with the use of dynamically loaded libraries, but this approach is not easy to use by our mod creators as they would be required to directly interface their mods with the C++ code of the game's engine. An alternative to this approach is to use more user friendly language embedded into C++ – e.g. Lua – and handle the interface between C++ and this language ourselves. This option also allows the execution of a code input to the game at runtime, which satisfies another of our requirements.

The reason we decided to choose C++ as the language our game is going to be written in is mainly a combination of the abundance of various game development related resources aimed at C++ – be it books, tutorials or even answered questions that can be found on internet – and the author's knowledge of the language.

2.2.2 Managed Languages: Java and C#

Managed languages, unlike native ones such as C++, are compiled to an intermediate language which can then either be interpreted by a virtual machine or JIT compiled into machine code of the target architecture. The use of a JIT compiler allows execution speeds that are comparable to those of C++.

Where they beat C++ is in their approachability, as they abstract memory management and other lower level aspects of programming from the programmer. As such, use of these languages would lead to an easier mod making process for our players, but as we have already settled in the previous section, that can be done in C++ using an embedded language with easier to understand syntax and semantics.

Both of the managed languages that were taken into consideration – Java and C# – provide an easy way to execute code input at runtime using either the Java Compiler API [20] or the Roslyn sompiler service [21] available in C#. Alongside this feature both languages offer the ability to embedd another language throu the use of libraries such as LuaJ [22] or NLua [23].

The difference of these two languages lies in their environment. The Java Virtual Machine – often abbreviated as JVM – provides the ability to compile the code once and then run the resulting executable file anywhere, which can be beneficial for an ordinary desktop application. When it comes to games, this advantage loses part of its strength due to the fact that the majority of the players use a Windows operating system – as can be seen in the Steam hardware and software survey [24]. According to the survey, over 95% of the players that use the Steam platform play their video games on a Windows system – ranging from Windows XP to Windows 10. In this case, the use of a virtual machine – be it JVM or .NET's Common Language Runtime – brings little to no advantage over a native language such as C++ in terms of portability. On the other hand, Java gains a disadvantage compared to C# as it requires the player to have the JVM installed, which is not installed on any of these operating systems by default. This means that our game would require the installation of a third party software – though the JVM can be bundled with the distributed game, it would still ask our players to install updates. The necessity of having the .NET platform installed is not a problem as it is developed by Microsoft, which is also the developer of the Windows operating systems, and as such can be installed through Windows and automatically kept up to date by the Windows Update service.

Because of this problem, the Java programming language was not chosen for the implementation of our game. C#, on the other hand, is an easier to use language than C++ and offers the ability to be modded in itself. Being able to have our game to be modded in C# would be beneficial seeing as the Unity3D game engine [25] uses it for scripting and thus many game developers and modders are already capable of using it. Considering these characteristics of the language, we find it to be equal – if not superior – to C++ in terms of game development capability. The reason for not choosing C# as the language to write our game in was the fact that the author has more experience with C++.

2.3   Scripting Language

Now that we have chosen C++ as the programming language we are going to implement the engine of our game with, we need to choose which language we are provide our modding interface in. Such a language should be easily embeddable within C++, easy to use and well known in the modding community so that people with modding experience can easily create mods for our game.

2.3.1   Lua

Lua is a programming language that was designed to be embedded into other languages like C or C++ and as such provides a simple to use API written in ANSI C allowing easy function binding and data sharing between C/C++ and Lua. These characteristics, along with others such as small memory footprint, easy

to understand syntax and high configurability using provided meta mechanisms, caused Lua to become the most favorite language used for game scripting [26].

Due to the high amount of games using Lua for scripting – e.g. the Wikipedia category called "Lua scripted video games" contains 157 entries [27] – there is already a large amount of mod creators that know how to use the language to create mods for games and as such the use of Lua in our game would make our modding tools more familiar to players that already have experience in modding.

2.3.2 Python

While Python is similar to Lua with its easy to understand syntax, its ability to be embedded to C++ is bit worse in comparison. The various C++/Python interface APIs are mostly designed to allow the extension of Python using C++ and as such require more work to embedd Python in C++ – e.g. unlike Lua, which uses a special stack to communicate with C++, the CPython API [28] requires manual reference decrementing and incrementing for heap allocated Python objects used in C++.

Where Python generally beats Lua is the abundance of libraries it has available, ranging from scientific libraries to image manipulation libraries. But since our scripting language will be mainly be used as an interface to the functionality of our engine, these libraries offer little to no advantage over Lua's minimalistic standard library.

The main downside of using Python as our scripting language is in the fact that Lua is used more often as a scripting language in games – the Wikipedia category called "Python scripted video games" contains 17 entries [29], which is much lower than Lua's 157 entries. This means that the modding community will probably not be used to writing mods in Python as much as they are in Lua. Additionally, Python uses the notion of syntactically significant whitespace – meaning that it uses whitespace to denote code blocks – which might be a bit confusing to a non-programmer that would want to mod our game.

2.3.3 AngelScript

AngelScript [30], similarly to Lua, is a programming language designed to be embedded into other languages for scripting. The main advantage it has over Lua is that it is even easier to embedd within C++ because of its C++-like design, requiring only a simple registration of C++ functions in order to be able to call them.

This advantage is also the main downside of the language, as its C++-like syntax is not as easy to understand as Lua's. This means that while AngelScript would be easier to embedd into the engine, the modding API wouldn't be as begginer friendly as if it were interfaced to Lua. Additionally, there is a smaller amount of games that use AngelScript for scripting– according to the official website [31], only 35 games use AngelScript for scripting. While this amount may be higher than the amount of games using Python for scripting, it is much lower than the amount of games that use Lua.

2.3.4 Conclusion

In terms of familiarity of the scripting language, Lua beats both of its competitors as it has been used for far more games and thus might be more known by the modding community. Additionally, the significant whitespace of Python and the C++-like syntax of AngelScript make us believe that Lua's syntax is the most begginer friendly and as such easier to understand by those of our players that are not programmers. Lastly, we find both Lua and AngelScript to be more easily embeddable into C++ when compared to Python.

Because of these reasons, we decided to choose Lua as our scripting language as it seems to be better than its competitors at most of our requirements.

2.4 Entity Representation

In this section, we are going to investigate different approaches for entity representation in our engine, that is, how is each entity – i.e. anything that is part of the game world, such as a minion, a wall, a trigger, a task or an event – will be structurally represented in our engine. This includes the entity's data, logic and relationships between different entities.

Because of our goals **(G2.1)**, which requires modifiability of entities, and **(G1.1)**, which requires our game to be performant, our requirements for the entity representation are:

- Extensibility: It should allow easy addition of new entity types so that mods can define new entities for their mods. It should, if possible, also allow this extensibility at runtime, which would provide means for easy runtime testing and prototyping.

- Modifiability: It should allow easy modification of predefined entity types so that mods can change entities that are already present in the unmodded game.

- Ease of Lua binding: It should be easily representable in Lua scripts.

- Performance: Since the entity updating will, along with rendering, take the majority of execution time, the representation should allow fast entity updates.

2.4.1 Inheritance

In inheritance based entity representation, an entity is a class. Characteristics of an entity are then implemented by inheriting from other classes and implementing interfaces. Since C++ does not allow the programmer to alter the inheritance hierarchy of the game without recompilation without the use of dynamically loaded libraries – which would require our modders to use C++ as this cannot be done through Lua – the modifiability of such entity is limited when compared to the modifiability the ECS representation provides.

While such an entity can be interfaced to Lua, Lua does not provide the object oriented paradigm by default and requires its implementation through the use of

meta mechanisms provided by the language. This means that even though the entities could be accessed from Lua, the modding API would have to be more complex – as simple function binding would not suffice – and would require the understanding of object oriented programming from our modders.

Unlike the Entity Component System option, which will be discussed next, this approach does not offer as good usage of the spatial locality of cache, because data are grouped by entity in memory. Additionally, the use of a virtual function is slower than a non-virtual one by about 25% [32] and also offers lower opportunities for inlining.

2.4.2   Entity Component System

Entity Component System[2] [33] [34] [35] – often abbreviated as *ECS* – is a structural design pattern that endulges the *composition over inheritance* principle. It comprises three main elements: *Entity*, *Component* and *System*.

- Entity is an identifier of anything that is present in the game world. It can be as simple as a numeric identifier or a more complex object, such as a component container.

- Component is a piece of logically related data, it generally represents a single characteristic of an entity, such as its health, position, collision box, movement or behavior.

- System updates a single component or a set of components of all entities that have these components.

In Listing 2, we can see an example of a system that – on a set period – regenerates the health of all entities that have a health component. The *health_system* iterates over all *health_components*, which contains all data related to health and regeneration. Updating the game state in ECS is then done by updating all systems.

This representation satisfies all of our four requirements. Since entities are nothing but a set of components, we can specify which types of components constitute an entity in a simple script and we can even create completely new types of entities during runtime by creating a new identifier and assigning components to it.

Similarly, we can add new components and remove existing components of an entity, which allows modification of already existing entities. As an example, we can add a movement component to a stationary entity to make it able to move. This, like the definition of new entities, can be easily done at runtime.

In both of these cases – modifying an existing or creating a new entity – ECS provides more flexibility than the inheritance based approach, because C++ does not allow us to change the inheritance hierarchy without recompiling the engine, while ECS allows any possible combination of components.

If we use numeric identifiers to represent an entity, we can easily interface our entities to Lua as all C++ functions called from Lua can take the identifier as their argument and then find necessary components in the component database.

---

[2]Sometimes also referred to as Component Entity System.

```
entity = {
    health_component = {
        health = 50,
        max    = 100,
        regen  = 2
    }
}

health_system = {
    components = { entity.health_component },

    function update()
        for comp in components do
            if comp.health < comp.max then
                regenerate(comp)
            end
        end
    end,

    function regenerate(comp)
        regenerated = comp.health + comp.regen
        comp.health = min(comp.max, regenerated)
    end
}
```

Listing 2: A simple health system that regenerates the health of every entity that has a health component.

This approach would avoid the necessity to implement object representation in Lua – which does not support object oriented programming by default.

An interesting characteristic of the ECS is its satisfiability of our last goal – performance. To store our components, we often use some kind of a key value associative container, where the key is the entity identifier and the value is the component. If our entity is not a numeric identifier, but a component container, it then contains pointers or references to the components stored in this cenral container. Since logically related data – components of the same type – are located next to each other in memory, we can then use spatial locality of cache to avoid some of the possible cache misses when we operate on components of a specific type, because the blocks of memory loaded to cache are less likely to contain data that are irrelevant to the current computation.

2.4.3 Conclusion

The ECS representation, unlike the inheritance based one, satisfies all of our requirements. It also offers greater modifiability and allows easy entity interfacing to Lua scripts and because of these traits was chosen for our engine.

2.5   Libraries

In this section, we are going to examine different libraries that can be used in our game to help us with its development.

2.5.1   3D Rendering

Since, according to our main goal, we are going to create a 3D game, the first library we need to choose is a 3D rendering library. Our game will, similarly to the original Dungeon Keeper, only have a relatively small amount of visible entities in the game world: minions, enemies, spells, buildings and walls. Because of this, we will settle with a relatively small functionality of the library. It will have to be able to load models created in a graphical editor so that our players can easily create new models and load them in their mods of our entities, render them to the game window and move them in the game world. In particular, the library should be compatible with models created in Blender [36], which is fairly popular, has a lot of documentation and is free, which means there will not be an additional cost of purchasing a graphics editor license to create models for our game.

Additionally, since our design element **(E4)** is the combat between our minions and enemies, support for animation [3] would be beneficial so that the combat is visible. Another of our design elements, **(E5)**, is the participation of the player in combat by the use of spells. This feature will require the ability to transform rendered objects to create spell effects such as explosions.

Another important aspect of a library is the ease of its use. Because of this, our chosen library should provide an easy to use interface, have complete and clear documentation and a large enough community.

Lastly, the library should be compatible with our ECS entity representation because the model of an entity is part of its data and as such will need to be part of one – or some – of its components.

Low level API: OpenGL and DirectX

Lower level API provides direct acces to the graphics hardware, common examples are OpenGL and DirectX. These interfaces, while extremely flexible, do not support models created in an external application and thus fail to satisfy one of our goals. Additionally, if we decided to use one of these interfaces, we would need to implement a lot of functionality – such as scene representation – that is already provided to us by a higher level library and thus would prolong the development time of our game. For these reasons, we would like to find a library that provides these features.

Higher level API: Ogre3D and Irrlicht

Another option is a higher level API, which is often a wrapper around a lower level API such as OpenGL and DirectX. The benefit of this option is that a higher level API often provides a large amount of already implemented functionality and

---

[3]Even though animation was planned for the game, it was not implemented because of time constraints.

data structures which a lower level API does not. For this decision, we retain all of our requirements for the choice of our rendering library and add an additional one – the higher level API should be able to wrap around both OpenGL and DirectX. The reason for this is that a common problem a user encounters when using a 3D application is related to drivers. Allowing the user to switch the underlying API used in case of program crashes might solve these problems [37].

We took two libraries into consideration – Ogre3D [38] and Irrlicht [39]. These were selected because they both have been previously used to create commercially successful games – Torchlight [40] and Octodad [41]: Deadliest Catch, respectively. Both of the considered libraries satisfy all of our requirements as they both support models created in an external application and their manipulation at runtime, provide complete and clear API documentations, both have large communities, wrap around both OpenGL and DirectX and provide an object representing a loaded model which can be integrated into a component in our ECS entity representation and can be easily interfaced to Lua. One difference between Ogre3D and Irrlicht is that an application that uses Ogre3D provides its user with a configuration window on startup, where the user can choose – along with other graphics options – which underlying API they would like to use. If we use Irrlicht, we would need to implement this configuration window ourselves.

Another difference between these two libraries is in the way they are integrated into our application. If we choose Irrlicht, we would need to create the game's main loop and call Irrlicht to draw our models. If, on the other hand, we use Ogre3D, the main loop is handled by Ogre3D and we only provide callbacks that update the game and handle events. While this difference might not seem significant, we think that the event handling done through different callbacks makes it easier to implement input handling and results into easier to read source code, since we do not need to check the type of an event and instead provide different handlers for different events – e.g. a key press, key release, mouse movement or mouse click.

These, albeit small, two differences were the reasons for choosing Ogre3D, since we think that these two libraries are equal on a technical level and the use of both would reach similar results.

2.5.2 Graphical User Interface

Now that we have chosen our 3D rendering library, we need to choose a library that provides tools to create a Graphical User Interface – often abbreviated as *GUI*. The main requirement for the library is its compatibility with Ogre3D as they will both need to render to the same window. Since this library will be used to create the interface our players will use to control the game, it has to offer widgets that will allow for control of all our design elements including, but not limited to:

- Button – needed for spell and building choosing and research.

- Label – needed to display current amounts of the game's resources.

- Frame – needed as a background for other widgets.

Aside of these three, the library should also contain widgets that would allow us to create in game console – that is an edit box, text box and a scroll bar – which would be beneficial for testing and prototyping. Lastly, to allow modifications of our GUI, an external GUI editor that can be used to create configuration files representing the GUI of our game would be beneficial.

Ogre Overlay

Overlays are a feature of the Ogre3D library which allows us to render 2D and 3D elements on top of the normal scene. It can be used to create windows, buttons, labels and other widgets but does not provide already prebuilt widgets. This would mean that we would need to implement all the widgets we need along with a system that handles the interaction between the user and our GUI.

While this option does not require any additional library to be used in our project, the development cost would be too high when compared to a GUI library that already provides these widgets as well as means for input handling. Because of this and the lack of an external GUI editor, we decided to use a dedicated GUI library in our game.

CEGUI

CEGUI [42] is a GUI library that was originally a part of the Ogre3D library and as such is fully compatible with it. The library provides a wide array of different widgets, including all of the one we require and many more.

To create our GUI schemas, we can use the Unified Editor for CEGUI [43], which generates XML files that contain information about our scheme such as the position, size or text of our widgets. Our mod creators can modify these XML files to modify our GUI.

Because CEGUI satisfies all of our goals and because it was previously bundled with Ogre3D – meaning there are tutorials, documentation and community discussions related to the integration of these two libraries – we decided to choose CEGUI as the GUI library for our game.

2.6   Pathfinding

An important aspect of most games is the ability of entities to find the way to their target inside the game world. This is done through a process called *pathfinding*. From a programming point of view, pathfinding is a process of finding – often the cheapest – path inside a pathfinding graph. Since our game, similarly to the original Dungeon Keeper, is tile based – meaning that the game world is divided into a set of adjacent square tiles, each representing a node in the pathfinding graph – our pathfinding graph will have the form of a grid. Nodes in this grid will be connected by edges to all of their neighbours. Nodes can be in the corner, on the side or inside the pathfinding graph and have three, five or eight neighbours, respectively.

We now need to decide which algorithm we are going to use for pathfinding. Since diagonal edges between nodes are visually longer than non-diagonal ones,

the algorithm needs to work on a pathfinding graph with weighted edges. Additionally, pathfinding will be performed quite often and as such our algorithm must try to avoid searching the entire pathfinding graph to find a suitable path.

The algorithms we took into account are the Dijkstra's algorithm, the breadth-first search and depth-first search algorithms – often abbreviated as *BFS* and *DFS*, respectively – and the A* algorithm. Out of these options, we decided to choose the A* algorithm as it works on weighted graph and uses a heuristic to avoid unnecessary searching. The BFS and DFS algorithms failed to satisfy our requirement that the algorithm should work on weighted graph and the Dijkstra's algorithm, while working on weighted graphs, does not use a heuristic to avoid examining some of the nodes and tends to run slower than A*.

2.7   Levels and serialization

To satisfy our goal (**G2.3**), our game has to provide the ability to create custom levels that can be distributed by players and then loaded by other players. Unlike the original Dungeon Keeper, which had a predefined set of levels, our game uses randomly generated levels. The primary reason why we chose this change is that we believe that randomly generated levels can extend the replayability of our game – since the amount of different levels is not limited to a small set. The secondary reason is the nature of our game – the player is the one that shapes the world by building their dungeon.

The impact of this decision is that we do not need to store our levels as they get created whenever the player starts a new game. Because of this choice, we are going to satisfy our goal (**G2.3**) through the means of serialization. By serialization, we mean saving of the game's current state to a persistent file on the hard disk which can later be loaded back into the game and the player can continue from the serialized point.

This way, if a player wants to create a custom map, they can generate a new level, modify it, serialize it and distribute the created file to other players that can easily load it. Since we want to allow our users to create custom levels similar to the Team Fortress 2 map created in Minecraft that was presented in the introduction of this thesis, our only requirements – besides saving the game's state – is the ability easily edit these serialized files and through these edits change the state and mechanics of the game.

2.7.1   Binary

During a binary serialization, the game state – in our case the components of entities and some additional information – gets serialized as a sequence of bytes into a binary file. While this option results in generally smaller save files than a text serialization, it prohibits modification of the game's mechanics. Additionally, even though this format allows our players to modify the serialized game state, these modification would require an application that would parse the save file, allow the user to change the data and then serialize the data again.

2.7.2 XML

Another considered option, the Extensible Markup Language – often abbreviated as *XML* – would allow us to serialize the state of our game to a human readable yet easily parseable format. This would allow our players to easily change characteristics of the serialized entities – e.g. their health, position or types of components – and even to add new entities or delete existing entities from the game state.

However, similarly to the binary format, XML files cannot contain custom code that would alter the mechanics of our game.

2.7.3 Lua

The last considered option is serialization into Lua scripts, which means that our game generates a sequence of calls to our Lua API that transforms an empty level to the serialized one and stores these calls into a source file that is then executed by Lua when the player wants to load the level.

This option allows our players to use the entire modding API inside these save files, which means that entire mods can be distributed in the form of a serialized game state which would satisfy our goal **(G3)**. Additionally, this feature is very easy to implement because of our entity representation. The game's state comprises the states of individual entities along with some additional information – e.g. state of the wave system, research and resources. This means that we only need to implement serialization of the different components in the game along with the aforementioned additional information using the Lua API. Deserialization does not need to be explicitly implemented as it constists of deleting the current game's state – if any – and executing a Lua script that represents the desired game state.

This option has one major downside when compared to the other considered options – the size of the scripts. With a very large world, the serialized state of the game becomes too big and can cause long loading times and thus limits the size of the game world. However, we do not consider this limitation too bad as we found smaller levels sufficient for general play.

2.7.4 Conclusion

While the first two options – binary and XML serialization – would allow us to serialize bigger worlds than we can with Lua scripts, the ability to edit both characteristics of the serialized entities and the mechanics of the game that Lua serialization provides us is, we think, worth the bigger size of the saved file when a large level gets serialized. Because of this, we decided to choose Lua script generation as our serialization system.

3. Developer's Documentation

In this chapter, we will examine the structure of our game and the implementation of its various modules. In Figure 3.1, we can see the main modules of our engine and their relationships – an arrow pointing from module A to module B means that A uses B.
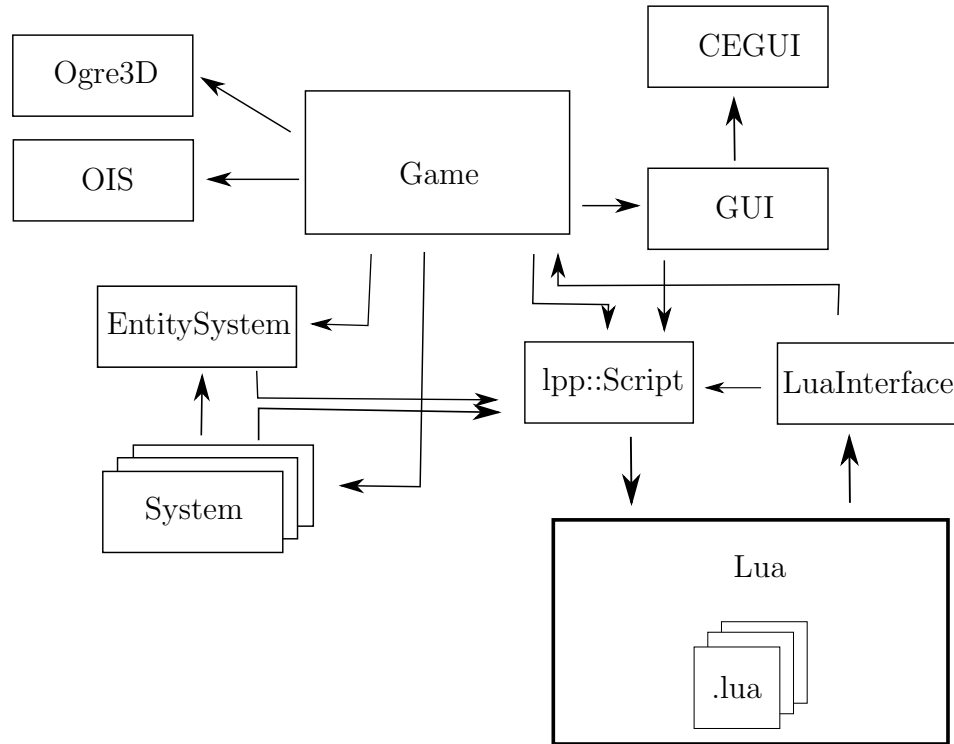

Figure 3.1: Core of the game's engine.

The game uses four different libraries – Ogre3D for graphics, OIS[1] for user input, CEGUI for GUI and Lua for scripting. Ogre3D and OIS are used directly by the engine, but CEGUI and Lua – besides initialization in the `Game` class – use special interface modules that handle their API. The GUI module, which acts as an interface for CEGUI, comprises the `GUI` class and various different classes that represent different GUI widgets. Lua uses different classes for the different directions of access. When the engine – written in C++ – accesses Lua, it uses the `lpp::Script` class, which acts as a wrapper facade over the Lua C API. When Lua accesses parts of the engine, it uses the `LuaInterface` class, which binds functions written in C++ to Lua.

The update logic of the game is performed by systems which, with the exception of `EntitySystem`, are equal in functionality. `EntitySystem` stands out because it, besides updating the game, aslo acts as a component manager and provides components to the rest of the systems.

In the following sections we will examine these main modules of the engine as well as various other, which mostly serve as tools to the main modules.

---

[1]Bundled with Ogre3D.

3.1 Game

The `Game` class is the central part of our engine. In its constructor, every module of the engine – that needs initialization – is initialized. This class also serves as the main connection between the Ogre3D and OIS libraries and our engine.It does so by inheriting four listener classes from these libraries – `Ogre::FrameListener`, `Ogre::WindowEventListener`, `OIS::KeyListener` and `OIS::MouseListener`.

```
// Ogre3D
bool frameRenderingQueued(const Ogre::FrameEvent&);
void windowResized(Ogre::RenderWindow*);
void windowClosed(Ogre::RenderWindow*)

// OIS
bool keyPressed(const OIS::KeyEvent&);
bool keyReleased(const OIS::KeyEvent&);
bool mouseMoved(const OIS::MouseEvent&);
bool mousePressed(const OIS::MouseEvent&,
                  OIS::MouseButtonID);
bool mouseReleased(const OIS::MouseEvent&,
                   OIS::MouseButtonID);
```

Listing 3: Virtual functions overriden in the `Game` class.

Communication between the game and these two libraries is done, besides using their API, by overriding virtual functions of these classes which act as event handlers. We can see these functions in Listing 3. While names of most of these functions are self-explanatory – and their documentation can be found in the Ogre3D and OIS documentations – we will examine the functionality of `frameRenderingQueued` in a bit more depth.

`Ogre::FrameListener` provides three virtual functions – `frameStarted`, which is called when a frame is about to begin rendering, `frameEnded`, which is called when a frame has just been rendered, and `frameRenderingQueued`, which gets called when all rendering commands have been issued and are queued for the GPU to process. In our `Game` class, we override only this last function – although others can be overriden for different actions as well – because while the GPU processes the current frame, the CPU can be used for different tasks – such as our update loop. This means that any changes that happen inside this function will be rendered on the next frame, but in our game this delay is not noticeable as the player does not directly control any one entity. We use this function to perform the main functionality of our game – performing game update by calling the update functions of all systems that are in the game.

Besides handling input and window events, this class contains several auxiliary functions related to initialization, level creation and state changing. To see more information about these functions, refer to their documenting comments in the header file or in the generated documentation.

3.2 Components

Component is a data aggregate that describes a specific characteristic of an entity that contains the component. In our engine, these components are represented as simple structures with a name in the form of `<Characteristic>Component`.

```cpp
struct MovementComponent
{
    static constexpr int type = 4;

    MovementComponent(tdt::real speed = 0.f)
        : speed_modifier{speed}, original_speed{speed}
    { /* DUMMY BODY */ }

    tdt::real speed_modifier;
    tdt::real original_speed;
}
```

Listing 4: Simplified representation of the MovementComponent structure.

In Listing 4, we can see an example of a component – the MovementComponent, which describes the speed of an entity. Every component has a static integer field representing its type, which is used for communication between Lua and C++ because Lua does not understand the notion of templates, which we use for the different functions that manipulate with components, and as such it needs to specify what kind of component should be used when it calls a C++ function.

There is only one other requirement for components, which is that they always need to have a parameterless constructor [2]. This is required because we often create components by calling a C++ function in Lua and then set its fields, which would be impossible without a constructor that doesn't need any parameters. Also note that because of this approach, any strings passed to any parameterized constructor are considered to be passed from Lua and as such are **moved** because the originals are supposed to be destroyed by Lua once the creation is complete.

Besides the two requirements, components can contain any data required by the entity characteristic they describe. In the examle, the MovementComponent contains two fields of type `tdt::real`[3] – speed_modifier, which indicates the speed of an entity, and original_speed, which is used to restore the speed of an entity that has been affected by a slowing effect. Note that, to conform our ECS representation, components should not contain functions and any logic should be done through systems.

3.3 Systems

A system is a class that performs a part of the games update. The game provides a common abstract parent class `System`, which is used for easy iteration over all

---

[2] A constructor that has default values for all its parameters will suffice.

[3] One of two numeric types used in the engine, the other being tdt::uint.

systems during the update of the game. In general, a system should operate over one or more types of components, but the creation of systems that do not use components is also possible.

The update function of every system is called once per frame, but the individual systems often have inner time periods between updates.

3.3.1 EntitySystem

`EntitySystem` is the main system as, besides performing part of the game's update, it acts as a component database. This means that it stores all component containers, which are represented as `std::map<entity_id, component>`. Besides storing the components, it also provides a set of function templates used for component manipulation.

```
template<typename COMP>
bool has_component(tdt::uint id);

template<typename COMP>
COMP* get_component(tdt::uint id);

template<typename COMP>
void add_component(tdt::uint id);

template<typename COMP>
void delete_component(tdt::uint id);
```

Listing 5: Examples of the templates using for component manipulation.

Some of these function templates can be seen in Listing 5. They can be used to test if an entity has a component, to retrieve a component that belongs to an entity, add a new component – using the parameterless constructor – to an entity and remove a component from an entity, respectively. In addition to these templates, each of them has a secondary non-templated version which takes an integer as a second parameter, which corresponds to the type of the component – these functions are then used from Lua. To prevent code repetition, these functions use an array of pointers to the templated versions in which the pointer at any given index points to the instances of these templated functions that have their type field equal to the index number.

In addition to these public templates, one more important private template is defined in `EntitySystem` – `load_component`. This function template accepts the identifier of an entity along with a name of a Lua table, which it will then use to load a component with fields specified by the provided table.

Besides component storage and manipulation, this class also serves as a system. During its update, it deletes all components and entities that were scheduled for deletion by the function `delete_component`. The reason for this delayed delete is that if we deleted components from their containers immediately in the delete call, we might invalidate iterators as the call may very well happen during an iteration over a component container.

3.3.2 CombatSystem

The `CombatSystem` class performs the update of basic combat between entities –
that is, melee [4] and ranged combat excluding spells. During its update, it checks
the ability to attack of all entities that currently have an active combat target.
This includes checking if the target is in sight and in range. If an entity can
attack, the system either performs applies its damage to its target if the attack is
of the melee type or creates a new projectile if the attack is of type ranged. After
updating the combat state of all entities that are currently fighting, the system
updates the movement of all projectiles.

This system also performs a special kind of pathfinding which is used to find
a path for an entity that is trying to escape from an attacker. This pathfinding,
unlike the general pathfinding that finds a path to the target, uses a queue and is
performed once per frame. The reason for this is that this pathfinding does not
need to be performed immediately, while the general pathfinding is often used to
check for the existence of a path and as such needs to be finnished right after its
start so that the return value can be used.

In addition to performing part of the game's update, `CombatSystem` provides
two important types of functions – querying for closest entity that satisfies a
condition, applying an effect to all entities in range that satisfy a condition.

Conditions and entity querying

Conditions are functors – that is, structures that overload the function call
operator – which are used to test if an entity satisfies a certain requirement.
`CombatSystem` provides a function template that can be used to find the closest
entity that satisfies a condition.

```
template<typename CONT, typename COND>
tdt::uint get_closest_entity(tdt::uint, COND&, bool);
```

Listing 6: Signature of the entity query function template.

In Listing 6, we can see the signature of this function template. Its template
parameters are `CONT`, which specifies the type of component container the func-
tion will be querying over, and `COND`, which specifies the condition functor. The
function takes the identifier of the entity that performs the query, instance of the
condition functor and a boolean value determining if the target has to be in sight
and returns the identifier of the closest entity that satisfies the condition.

Effects and their application

In addition to entity querying, `CombatSystem` provides a function template that
allows us to apply an effect to all entities that satisfy a condition and are withing
range. An effect, similarly to a condition, is a functor which is used to affect the
entity it is used on.

The signature of this function template can be seen in Listing 7. Its template
parameters are the same as the ones used in entity querying with the addition of

---

[4]By melee we refer to close range combat as is common in many games.

```
template<typename CONT, typename COND, typename EFFECT>
void apply_effect_to_entities_in_range(tdt::uint, COND&,
                                       EFFECT&, tdt::real);
```

Listing 7: Signature of the effect applying function template.

the parameter `EFFECT`, which specifies the effect functor. The function takes the identifier of the entity that applies the effect, the condition the targets need to satisfy, the effect that will be applied to the targets and the maximal range the targets can be from the applying entity.

3.3.3 EventSystem

`EventSystem` manages event handling as part of the game update. To do this, it uses two different components – `EventComponent`, which represents an event, and `EventHandlerComponent`, which represents the ability of an entity to handle events.

During its update, this system iterates over all events in the game and, if they are active – finds a suitable entity that will handle the event. Events can be of two types, either targeted or area events. An event is targeted if it has a valid entity identifier in its `handler` field, otherwise it is an area event.

To request an entity to handle an event, the system checks if the entity can handle it by testing the event type in the entity's `possible_events` bitset field. Every `EventHandlerComponent` has a string field `handler`, which specifies which Lua table contains the event handling function called `handle_event`. The system then calls this event handling function and passes the identifier of the handling entity and the identifier of the event to it, which causes the entity to handle the event.

To handle a targeted event, the system simply finds the event's handler entity and requests event handling, but to handle an area event, the system has to iterate over different event handlers that are within radius of the event, which is specified in the `EventComponent`'s field `radius` and increases on every update call until the event is destroyed. The system requests all of the suitable entities to handle the event until any one entity returns `true`, in which case it destroys the `EventComponent` of the event. The reason for destroying the component and not the entity that represents the event is that an entity that causes the event often gains the component so that the lifetime of the event is bound to the entity – e.g. a falling meteor can have an `EventComponent` of type `METEOR_FALLING` and when the meteor hits the ground, both the meteor and the event get destroyed since they are one entity.

3.3.4 TaskSystem

Similarly to how `EventSystem` manages event handling, `TaskSystem` manages task handling. It does so by iterating over all entities that have `TaskHandlerComponent` and calling the `handle_task` Lua function that is located in the Lua table which is specified in the component's field `blueprint`. This Lua table also contains

a function called `task_complete`, which is used to check if the current task an entity that is iterated over has been completed, in which case the system requests handling of the next task in the component's `task_queue`.

Additionally, an entity that is handling a task can enter a busy state by returning `true` from its task handling function. While an entity is marked as busy, no other task handling nor any AI updates will be performed until its current task is completed.

3.3.5   GridSystem

`GridSystem` manages nodes in the pathfinding graph and structures that are placed on them. During its update, it examines all nodes that were either freed, which means that a structure on them has been destroyed, or unfreed, which means that a building was placed on them. The system uses lists of freed and unfreed nodes to correct paths that were blocked by a strucute placed on a node that was part of the path. Additionally, it manages all entities that have `AlignComponent`, which is used to change the models of walls depending on their neighbours.

Grid

`Grid` is an auxiliary singleton class that provides an interface for grid node manipulation and monitoring, which acts as a wrapper around a set of entity identifiers that belong to the different nodes in the pathfinding graph. Its main use are the lists of freed and unfreed nodes that are used by `GridSystem`. In addition to these lists, `Grid` provides functions for graph creation, placement of an entity on a random node and distribution of multiple entities on adjacent nodes – which can be used to spawn a group of entities next to each other.

3.3.6   WaveSystem

`WaveSystem` manages the progression of enemy waves that attack the player's dungeon. During its update, it tracks time passed since the last enemy wave and starts the next wave if needed. The system can be in three different states – `ACTIVE`, `WAITING` and `INACTIVE`. When the system is in the `ACTIVE` state, an enemy wave is currently attacking the player's dungeon and the system spawns enemies on spawning nodes that were selected during level generation. After the player's minions destroy all enemies in the current wave, the system transitions into the `WAITING` state. During this state, the system simply tracks a timer until the next wave. If the system depletes all waves or is paused, it goes into the `INACTIVE` state in which it waits until the wave is either resumed or restarted.

To monitor the current state of the wave when the system is in the `ACTIVE` state, every spawned entity increments the entity counter and the system adds `DestructorComponent` to it which specifies which function should be called when the entity dies. This function then decrements the entity counter. When the counter reaches zero and all entities have been spawned, the wave ends and the system transitions to either `WAITING` or `INACTIVE` state.

A specific sequence of waves is defined by a Lua table. The name of the current wave Lua table is stored in the variable private `wave_table_`. This table contains

functions that prepare the wave sequence and start and end the individual waves. This table, along with a tutorial on how to create a new wave sequence can be found in Chapter 4.

3.3.7 Miscellaneous Systems

Now that we have examined the bigger systems that are included in the game, we will briefly explain the update logic of the remaining systems, which generally perform simple tasks.

- `AISystem` iterates over all entities that have `AIComponent` and calls the Lua functions that update their behavior if they are not currently performing a task.

- `GraphicsSystem` was designed to manage all manual graphics. Explosions are the only manual graphics that are currently in the game so the system updates the dimensions of all entities that have `ExplosionComponent` and destroys them once they reach their limits.

- `HealthSystem` removes all entities that are marked as not alive and periodically regenerates their health.

- `InputSystem` handles the input from the player and uses it to control all entities that have `InputComponent`.

- `ManaSpellSystem` regenerates mana of entities that have `ManaComponent` and manages the spell casting of all entities that have `SpellComponent`.

- `MovementSystem` manages the movement of all entities that have `MovementComponent` and are currently on a path. Note that this excludes homing projectiles, which are updated by `CombatSystem`.

- `ProductionSystem` manages the production of entities by buildings. It tracks time periods between spawns and the spawn limit of a building and when the time comes and the spawn limit is not depleted, the system spawns a new entity either on the building if its just a tile on the ground or around the building if its solid and cannot be stood on.

- `TimeSystem` manages the different timers in the game, including, but not limited to the timers of `OnHitComponent`, `LimitedLifeSpanComponent` and `TriggerComponent`. Additionally, it manages a special kind of events that are represented by `TimeComponent`. These components can start or end events represented by `EventComponent` or call Lua functions.

- `TriggerSystem` manages entities with `TriggerComponent` such as traps or portals. It checks all entities in range to see if they can trigger the entity with `TriggerComponent` and if they can, calls the trigger function.

Ask PJ: Should the `SomethingComponent` have articles?

3.4 lpp::Script

The `Script` singleton class in the namespace `lpp` is a wrapper facade around the Lua C API which provides an easy to use interface for communication between C++ and Lua. It provides functions that allow us, for example, to execute Lua code passed as a string, register C++ functions to Lua, retrieve values from Lua and call Lua functions from C++. In addition to this wrapper, the `lpp` namespace contains the `Exception` class. Besides the conventional `what` function, which returns a string describing the nature of the exception, this class also has the `lua_what` function, which returns the text of the Lua error that occured.

3.5 LuaInterface

`LuaInterface` is a static class that contains functions that are part of our modding API and is used for simple initialization of the API. The reason for it being static is that Lua does not understand the notion of a function bound to an instance of a class and as such can bind only static functions.

During the initialization of our API, arrays of pairs of function names and function pointers are registered in Lua. Each of these arrays is then represented as a Lua table which contains the registered functions. To register a function pointer, it is required by Lua to have the signature of `int (*name)(lua_State*)`. Because of this, `LuaInterface` contains static functions with this signature that act as wrappers of the actually called functions.

```
int lua_wrapper_of_F(lua_State* L)
{
    tdt::uint param2 = GET_UINT(L, -1);
    tdt::real param1 = GET_REAL(L, -2);

    int result = F(param1, param2);
    lua_pushinteger(L, result);

    return 1;
}
```

Listing 8: An example of our modding API function implementation in C++.

Listing 8 contains an example of such wrapper. In this wrapper, we first need to retrieve the actual parameters for our functions. These parameters are located on the Lua stack – i.e. in reversed order – and we can use one our five macros for easy parameter retrieval defined in the `LuaInterface` source file. In this example, we use `GET_UINT` and `GET_REAL`, each of which takes a pointer to the Lua state and an offset from the top of the stack. Once we retrieve these parameters, we can call the wrapped function and push its result back onto the stack. Note that Lua supports multiple results from a function call and thus we can push multiple values onto the stack. The amount of returned results is then returned from the wrapper.

3.6   Helpers

Helpers are auxiliary functions that are used for component manipulation that are placed within a namespace related to a specific component – e.g. the `HealthHelper` namespace contains functions that manipulate `HealthComponent`. These functions are mainly in the form of a setter, which changes a field of the component after checking the validity of the new value, or getter, which returns the value of a gield of the component or a default value if the component does not exist. But the body of these functions is not restricted and they can perform any operation.

The main purpose of these functions is keep the `LuaInterface` class small, but can be used freely in C++ as well if we need for example parameter checking, default values or transaction logging. However, these functions generally retrieve the component from its container and as such sequential calls to different helper functions of a single component will be slower than a direct manipulation of the component.

3.7   SpellCaster

`SpellCaster` manages player spell casting. Player spells in our game can be of four different types – `TARGETED`, which affect a single selected target, `POSITIONAL`, which affect a specific area the player clicks at, `GLOBAL`, which have a global effect on the game, and `PLACING`, which place a new entity into the game.

The spell casting process has two phases. Firstly, it selects the spell type and spell Lua table name and calls an initializating function that is contained within the spell table. Secondly, it calls the casting function and passes parameters to it based on the spell type when the player clicks on the screen to cast the spell.

This process is strongly connected to Lua, the `SpellCaster` class only manages spell changing and the two described phases, but the effects of the spells are fully defined in Lua. In Chapter 4, we will examine the spell table and go over the spell creation process.

3.8   Pathfinding

In Chapter 2, we decided to use the A* algorithm for pathfinding in our game. While A* is the implemented algorithm, the game contains a templated pathfinding system using functors for algorithms, heuristics and path types. This allows us to easily change the characteristics of our pathfinding process.

The `pathfind` function calls the algorithm functor, which has a static function `get_path` that returns the found path. After obtaining the path, `pathfind` resolves blocks that are placed on the path by requesting the entity that performs the pathfinding to destroy them before continuing on its path. The algorithm functor is templated as well and its template parameter is what we call a path type.

Path type is a functor that has a static function `return_path`, which is called the first time a path is found and then whenever an augmenting edge is found on the path. If the `return_path` returns `true`, the path gets returned, and if it returns `false`. Examples of such path types are the three path types that are implemented in the game – `BEST_PATH`, which forces the algorithm to search the

entire pathfinding graph, `FIRST_PATH`, which terminates the algorithm the first time a path is found, and `RANDOM_PATH`, which acts as a compromise between the two aforementioned path types.

Lastly, both the `pathfind` function and the algorithm's `return_path` function take a heuristic as a parameter. The heuristic, unlike algorithm and path type, is not static as it in some cases requires to have a state – for example, the heuristic used for running away from an enemy needs to know the identifier of the enemy. The heuristic has a function `get_cost`, which returns the approximate cost of travel between two nodes in the pathfinding graph.

3.9   Player

`Player` is a singleton class primarily used for resource monitoring – it stores information about the amount of all the player's resources such as gold, mana, spawned entities and others. Additionally, it contains information about the starting state of the game, such as initial spells and unlocked research. This is mainly used when we create a new game or when we load a previously saved game.

3.10   Serialization

The `GameSerializer` class is used to save the state of the game and to load previously saved states. It does so by creating Lua scripts that use our API to change an empty level to the serialized one. For every entity, it iterates over all of its components and outputs a sequence of API calls that add the component to the entity and restore the values of its fields. For this, the class has an array of pointers to functions that serialize the different components indexed by the types of the components.

In addition to the component serialization, it also saves resources that are contained in the `Player` class, creates an empty level, recreates the pathfinding graph, saves unlocked spells and buildings and the overall progress of the player's research.

To load a saved state of the game, it simply destroys all the entities in the game and resets the `Player` class to its initial state. It then executes the serialized script to restore the state of the game.

3.11   Tools

In this section, we will examine some of the smaller tools that are used within the engine of our game.

Camera

The `Camera` class acts as a wrapper around the main `Ogre::Camera` instance. Its main purpose is to allow us control the camera in multiple ways – using both keyboard and mouse – and it also allows us to change between the basic mode, in

which the camera has a fixed orientation, and the free mode, in which the player can move and rotate the camera freely in the game world.

EntityPlacer

`EntityPlacer` is used to place entities in the game world. To place an entity, we first need to call the function `set_current_entity_table`, which will change the entity that is currently being placed and show its model at the position of the mouse cursor. While an entity is being placed, the game will call the function `update_position` which will move the temporary model to the specified position, which is used to make the temporary model to follow the mouse cursor. Once the player clicks on a spot in the game world, the game calls the function `place`, which will spawn a new instance of the entity that was placed at the current position of the placer.

LevelGenerator

Level generators, found in the namespace `level_generators` are used to populate an empty game world with starting structures such as buildings, walls and gold blocks. During the initialization of our game, the `Game` class creates an instance of the generator that is defined – using either `typedef` or `using` – as `DEFAULT_LEVEL_GENERATOR` in this namespace.

This forces the constructor of any level generator to take a reference to the `EntitySystem` class and a `tdt::uint` because the instantiation of the level generator is done generically with these two parameters. The reference to `EntitySystem` is used for the creation of walls, blocks and others and the unsigned integer denotes the amount of iterations of the generation – if the level generator is iterative, otherwise this parameter may be ignored.

Any level generator has to also have a function with the signature `void generate(tdt::uint, tdt::uint, WaveSystem&)`. This function will then generate a level with dimensions equal to the passed integers and sets the spawn points for the passed wave system.

RayCaster

The Ogre3D library provides only a basic bounding box ray casting, but our walls do not have to fill their entire bounding box because of alignment. For this, we used external code from the official Ogre3D website [44] in our `RayCaster` class because of the author's limited knowledge of graphics programming. This class is used to check if two entities can see each other by creating a ray between two points in the game world which, unlike standard Ogre3D ray casting, takes only actual models into account. The class uses this ray to determine if there is a block between the two entities and if so, assumes they cannot seee each other because of this block.

TODO: Add a screenshot when I get on windows and show the problem on it!

SelectionBox

`SelectionBox` is a class used for entity selection using either the plane bounded volume query or ray casting provided by Ogre3D. Whenever a player clicks on anything in the game world – besides parts of the GUI – the `Game` class starts selection by setting the initial point of the selecting square. Any mouse movement then sets the end point of the square so that the player has a visual feedback on his selection. Once the player releases his mouse button, the query is executed on the created selection square to select all entities that are located within it.

This class provides two modes of selection – single selection and multi selection. The type used is determined by the size of the selection square, so if the square is too small, a ray cast will be used instead of a volume query.

3.12 GUI

Our GUI module serves as a set of wrapper classes around CEGUI windows. The `GUI` singleton class is used for initialization and shared control of all other windows – e.g. to recursively toggle visibility of the windows. This class also contains instances of most[5] of the other windows. Each of these windows inherits from `GUIWindow`, which provides common interface for visibility and initializaion.

The game's GUI contains the following windows:

- `Console` – accepts Lua commands as its input and executes these commands in the Lua virtual machine.

- `EntityCreator` – a window that allows the user to spawn entities by choosing them from the list of all available entities, mainly used for testing.

- `EntityTracker` – display characteristics of the currently selected entity, e.g. its health or mana.

- `ResearchWindow` – button based interface for research, the player can click on a research node if they have enough resources to unlock it.

- `BuilderWindow` – provides the player with a selection of building they can build.

- `SpellCastingWindow` – provides the player with a selection of spells they can cast.

- `GameLog` – shows game messages, e.g. error messages when the player does not have enough resources to build a building.

- `MessageToPlayerWindow` – a simple multi-purpose window with a message and buttons.

- `OptionsWindow` – allows the player to change the game's resolution, toggle fullscreen mode and change key bindings.

- `TopBar` – a small bar at the top of the screen that shows the player's current resources.

---

[5]It does not contain some testing windows, such as `EntityCreator`.

Bibliography

[1] Orcs must die! `http://en.omd.gameforge.com/`. [Online; accessed 2016-05-15].

[2] Dungeon Defenders. `https://dungeondefenders.com/1/`. [Online; accessed 2016-05-15].

[3] Tuur Ghys. Technology Trees: Freedom and Determinism in Historical Strategy Games. `http://gamestudies.org/1201/articles/tuur_ghys`. [Online; accessed 2016-05-17].

[4] Minecraft. `http://www.minecraft.net`. [Online; accessed 2016-04-23].

[5] Curse Minecraft Mod Repository. `http://mods.curse.com/mc-mods/minecraft`. [Online; accessed 2016-06-14].

[6] PlanetMinecraft Minecraft Mod Repository. `http://www.planetminecraft.com/resources/mods`. [Online; accessed 2016-06-14].

[7] Industrial Craft 2. `http://www.industrial-craft.net/`. [Online; accessed 2016-05-17].

[8] ComputerCraft. `http://www.computercraft.info`. [Online; accessed 2016-05-17].

[9] Lua. `http://www.lua.org`. [Online; accessed 2016-05-09].

[10] PCGamer. Future of Minecraft. `http://www.pcgamer.com/the-future-of-minecraft/`, 2012. [Online; accessed 2016-04-23].

[11] Team Fortress 2. `http://www.teamfortress.com`. [Online; accessed 2016-04-23].

[12] M. F. Shiratuddin, K. Kitchens, and D. Fletcher. *Virtual Architecture: Modeling and Creation of Real-Time 3D Interactive Worlds*. Lulu Press, 2008.

[13] UnrealEngine 4. `https://www.unrealengine.com`. [Online; accessed 2016-06-28].

[14] Starbound. `http://www.playstarbound.com`. [Online; accessed 2016-05-09].

[15] Roberto Ierusalimschy. *Programmin In Lua*. 3rd edition. Lua.org, 2013.

[16] Cities: Skylines. `http://www.citiesskylines.com/`. [Online; accessed 2016-07-09].

[17] Factorio. `http://www.factorio.com/`. [Online; accessed 2016-07-09].

[18] J. Gregory. *Game Engine Architecture*. 1st Edition. A K Peters/CRC Press, Boca Raton, FL, 2009.

[19] Garage Games. Why Use Scripting, Torque Engine documentation. `http://docs.garagegames.com/tgea/official/content/documentation/Scripting%20Reference/Introduction/Why%20Use%20Scripting.html`. [Online; accessed 2016-05-10].

[20] Java Compiler API Documentation. `http://docs.oracle.com/javase/8/docs/api/javax/tools/JavaCompiler.html`. [Online; accessed 2016-06-28].

[21] Roslyn Repository. `https://github.com/dotnet/roslyn`. [Online; accessed 2016-06-28].

[22] LuaJ Java Library. `http://www.luaj.org/luaj.html`. [Online; accessed 2016-06-28].

[23] Nlua .Net Library. `http://nlua.org/`. [Online; accessed 2016-06-28].

[24] Steam Hardware And Software Survey. `http://store.steampowered.com/hwsurvey?platform=combined`. [Online; accessed 2016-06-27].

[25] Unity3D Game Engine. `https://unity3d.com/`. [Online; accessed 2016-06-28].

[26] Mark DeLoura. The Engine Survey. `http://www.satori.org/2009/03/the-engine-survey-general-results/`. [Online; accessed 2016-07-02].

[27] Wikipedia Category: Lua scripted video games. `https://en.wikipedia.org/wiki/Category%3aLua-scripted_video_games`. [Online; accessed 2016-07-03].

[28] CPython C API. `https://docs.python.org/2/c-api`. [Online; accessed 2016-07-03].

[29] Wikipedia Category: Python scripted video games. `https://en.wikipedia.org/wiki/Category:Python-scripted_video_games`. [Online; accessed 2016-07-03].

[30] AngelScript. `http://www.angelcode.com/angelscript/`. [Online; accessed 2016-07-03].

[31] Games scripted in AngelScript. `http://www.angelcode.com/angelscript/users.html`. [Online; accessed 2016-07-03].

[32] Bjarne Stroustrup. *The C++ Programming Language*. 4th edition. Addison-Wesley, 2013.

[33] Entity Component System on Wikipedia. `https://en.wikipedia.org/wiki/Entity_component_system`. [Online; accessed 2016-07-09].

[34] Scott Bilas. A Data Drive Game Object System. `http://scottbilas.com/files/2002/gdc_san_jose/game_objects_slides.pdf`. [Online; accessed 2016-07-04].

[35] Entity Component System on Gamedev.net. `http://www.gamedev.net/page/resources/_/technical/game-programming/understanding-component-entity-systems-r3013`. [Online; accessed 2016-07-09].

[36] Blender. `https://www.blender.org/`. [Online; accessed 2016-07-05].

[37] Irrlicht features. `http://irrlicht.sourceforge.net/?page_id=45#drivers`. [Online; accessed 2016-07-05].

[38] Ogre3D library. `http://www.ogre3d.org/`. [Online; accessed 2016-07-05].

[39] Irrlicht library. `http://irrlicht.sourceforge.net/`. [Online; accessed 2016-07-05].

[40] Torchlight. `http://www.torchlightgame.com/`. [Online; accessed 2016-07-09].

[41] Octodad: Deadliest Catch. `http://octodadgame.com/octodad/dadliest-catch/`. [Online; accessed 2016-07-09].

[42] CEGUI library. `http://cegui.org.uk/`. [Online; accessed 2016-07-05].

[43] CEED: Unified Editor for CEGUI. `https://martin.preisler.me/2012/09/unified-editor-for-cegui/`. [Online; accessed 2016-07-05].

[44] Ogre3D: Raycasting to the polygon level. `http://www.ogre3d.org/tikiwiki/Raycasting+to+the+polygon+level`. [Online; accessed 2016-07-14].

Attachments