UNIVERSITY OF TARTU
Institute of Computer Science
Informatics Curriculum

Oliver Vinkel

Fall – A Turn-based Tactical Role-playing Game on a Hex Map

Bachelor's Thesis (9 ECTS)

Supervisor: Raimond-Hendrik Tunnel, MSc

Tartu 2019

Fall – A Turn-based Tactical Role-playing Game on a Hex Map

Abstract:

Fall is a video game created for this thesis and it incorporates the hex map in its game design in a way that was found to be unique amongst role-playing games. In addition to describing the implementation, the thesis discusses the use of regular tessellations in video games, compares *Fall* to other hex-based games and finally analyzes the results of player testing conducted of a small group of people to determine the viability of the concept of the game.

Keywords: hex map, regular hexagon, regular tessellation, video game, Unity game engine, computer graphics, computer games development, turn-based strategy

CERCS P170: Computer science, numerical analysis, systems, control

Fall – Käigupõhine taktikaline rollimäng kuusnurksete mänguväljadega kaardil

Lühikokkuvõte:

Fall on käesoleva lõputöö raames loodud videomäng, mis implementeerib heksipõhise kaardi rollimängude seas unikaalsel viisil. Lisaks mängu implementatsiooni kirjeldusele annab käesolev lõputöö ülevaate korrapäraste tessellatsioonide kasutusest videomängudes. Edasi võrreldakse loodud mängu teiste heksipõhiste mängudega ning lõpuks analüüsitakse väiksel grupil läbi viidud testimise tulemusi ning hinnatakse selle põhjal mängu idee potentsiaali.

Võtmesõnad: heksipõhine kaart, korrapärane kuusnurk, regulaarne tessellatsioon, videomäng, Unity mängumootor, arvutigraafika, arvutimängude loomine, käigupõhine strateegia

CERCS P170: Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Sisukord

1 Introduction

As more video games incorporating *hex maps*[1] are being released, it is a phenomenon that is continuously growing in popularity in the video game industry. While most commonly seen in video games developed by smaller development studios, hex maps have also been popularized by some hugely successful strategy games such as Endless Legend or Sid Meier's Civilization V. The latter game has received widespread critical acclaim and at the time of writing is owned by between 5 million and 10 million users[2] on the digital products distribution platform Steam[3] and despite being released many years ago, still maintains an active playerbase.

The aforementioned games are examples of *turn-based strategy games*[4]. Unfortunately there are no examples of hex-based *role playing games*[5] that have garnered a fanbase that large, even though this genre of video games has potential for providing entertainment value to a large audience as well. *Fall* is a game implemented for this thesis and while it is not ready to be released as a fully featured game, it explores an avenue of hex-based role playing games that was found to be either very rare or entirely unexplored. Hex maps in role playing games are usually limited to arena combat scenarios. *Fall* integrates the hex map with the whole gameplay experience and serves as an experiment to estimate how this subgenre of hex-based role playing games might be received by both casual and avid gamers. The distinctive features are listed in Appendix III.

Chapter 2 offers a brief overview of the different categories of tessellations, but focuses specifically on regular tessellations. Hex maps usually operate on a regular hexagonal tessellation and because this tessellation is an integral component of the implemented game map in this thesis, it will be explored in greater detail. All the while the context of how these tessellations can and have been used in video games is considered.

---

[1] https://en.wikipedia.org/wiki/Hex_map

[2] https://steamspy.com/app/8930

[3] https://store.steampowered.com/

[4] https://en.wikipedia.org/wiki/Turn-based_strategy

[5] https://en.wikipedia.org/wiki/Role-playing_video_game

Chapter 3 continues by categorizing various hex-based games based on similar traits and compares them with *Fall*.

Chapter 4 on the topic of implementation and architecture explains how various components of Fall were made and how the systems interact with each-other. The implementation described here serves as a guideline for other developers who may want to implement a similar game in the future. The particular needs of *Fall* may be specific to this thesis, but the chapter offers value to developers by providing example solutions for a potentially larger project.

Fall was also playtested with a group of people who supplied feedback on the various systems in the game. Their feedback was evaluated and playstyle observed. The process, results and conclusions are all provided in **Chapter 5**.

The game will continue to be developed after this thesis. Access to the project repository is reserved for the author, instructor and reviewer(s) of this thesis, but the Unity project files and a working build are found in the Appendix and are publicly accessible to any interested parties. Installation instructions are also provided there.

2 Tessellation

According to the book *Dictionary of Mathematics* [1], the term *tessellation* is said to have been originally used to describe the pattern formed by covering a planar surface with congruent squares (*tessara*), but the definition has nowadays been extended to also describe patterns formed by various other shapes as well. The book classifies the patterns as homogeneous (semi-regular), non-homogeneous and regular tessellations. Non-homogeneous tessellations and semi-regular tessellations have been used in video games before. See ***Europa Universalis IV*** in Appendix I for an example use of a non-homogeneous tessellation (Figure 48)[6] and ***Rust*** for an example of a homogeneous tessellation (Figure 49)[7]. Strategy games utilizing game maps commonly implement regular tessellations for the visual layout of the game map. Therefore the following subchapters will compare regular tessellations to describe the benefits and downsides of each type of regular tessellation. Furthermore, while any regular tessellation can be three-dimensional, this chapter focuses specifically on the characteristics of *planar* regular tessellations.

2.1 Regular Tessellations and Maps

Regular tessellation is the arrangement of one of three types of regular two-dimensional polygons in a mosaic pattern that does not include spaces in-between the polygons. There are no more than three regular polygons that can form a regular tessellation: equilateral triangles, squares and hexagons[8] (see Figure 1).

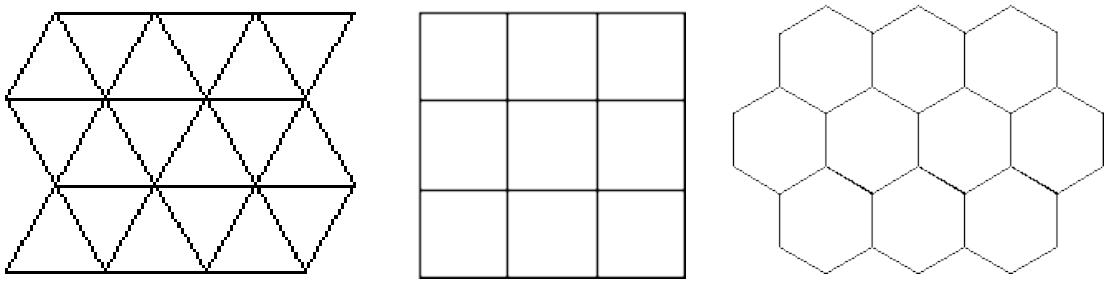

Figure 1. Regular tessellations (equilateral triangle, square, regular hexagon)

The main difference in the context of their use in video games is the number of neighbours each polygon has and how it affects gameplay.

[6] https://imperium.news/europa-universalis-iv-mod-roundup/

[7] https://youtu.be/GAh5klaaFck

[8] http://mathworld.wolfram.com/RegularTessellation.html

2.1.1 Neighbours

In a regular tessellation, an *n*-sided polygon can have at most *n* neighbours with which the polygon shares an edge and the vertices forming that edge. A triangle can have three neighbours, a square four and a hexagon six. In games using game maps the edges of the polygons (from now on *cells*) are usually involved in the gameplay. For instance they can be a part of the motion of game objects, where one object will transition over the edge to reposition itself onto another cell. In this case having more edges (neighbours) offers more directions for movement and of the three polygons hexagons provide the highest number of options. Squares can be a special case if diagonal movement is allowed, in which squares would provide a total of 8 directions with diagonal movement transitioning over the corner vertices of the origin cell. See Figure 2 below.

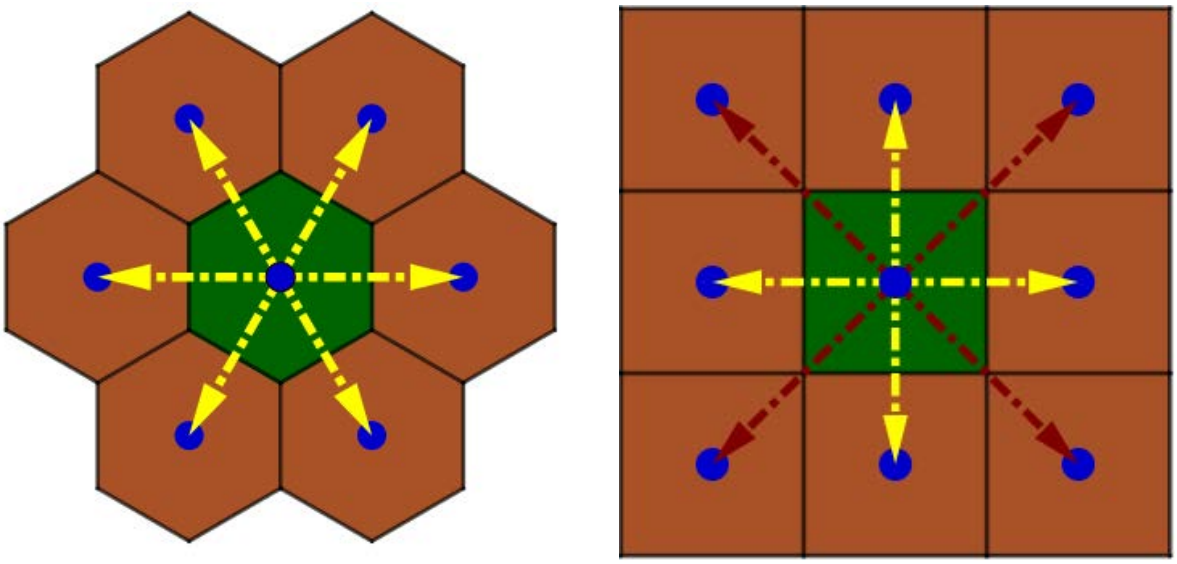

Figure 2. Transitions on hexes and squares

2.1.2 Equilateral Triangle Tessellation

In 3D graphics **triangles** are commonly used in the generation and rendering of complex surfaces [2]. Triangles are rarely used in game maps. The edges of the cells in game maps generally have a thickness value. Given a thickness, the edges of triangles would occupy a larger surface area than the edges in the other regular polygons under observation in this chapter. Additionally, given some edge length, the surface area of an equilateral triangle is smaller than the surface area of a square or a regular hexagon with the same edge length. As a result, the triangle tessellation will appear more compact in comparison. Maps in virtual game worlds are usually not required to be compact, therefore when selecting a polygon to use for the tessellation, other benefits are considered instead.

Additionally the neighbouring triangles in this tessellation are rotated at two opposing angles. This means that not every cell is the same and this has to be taken into consideration when designing a game based on this tessellation. Transitions on triangles can also become an issue if vertex transitions are allowed. Distance variation in edge transitions vs vertex transitions are depicted on Figure 3. Triangle tessellations are rarely used in game maps: squares and hexagons are much more popular.

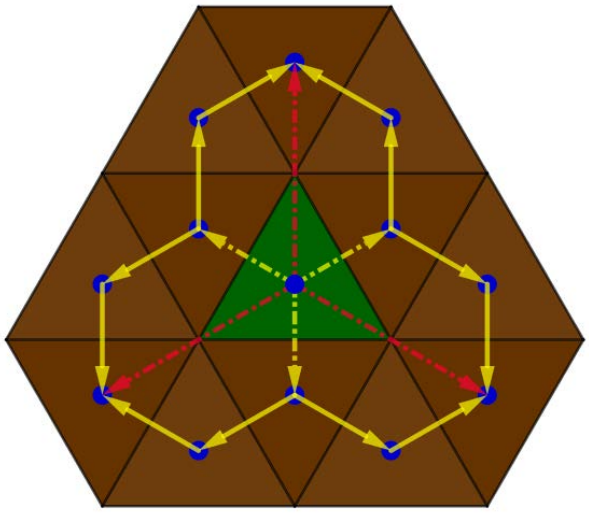

Figure 3. Transitions on triangles

It should be noted that the regular triangle tessellation is a subdivision of the regular hexagon tessellation[9]. A hexagon cell can be converted into triangles by inserting a vertex in the middle of a hexagon and connecting edges from that vertex to all six vertices of the hexagon.

2.1.3 Square Tessellation

In maps based on regular square tessellations, each square (or cell) is identifiable with a coordinate consisting of two arguments, *x* and *y*. The coordinates are written as (*x*, *y*), where *x* describes the column and *y* the row the cell is located in. Alternatively, rather than referring to the cell itself, the coordinates can refer to the edges or vertices in the tessellation[10].

Squares are computationally cheaper than the other options[11]. Square maps operate on two coordinates and if the motion of game characters is limited to horizontal or vertical movements, the distance from one cell to another cell is calculated as the *Manhattan distance*[12]. The sum of the absolute values of the change in coordinate arguments: $distance = |x_2 - x_1| + |y_2 - y_1|$ Algorithms for finding the nearest neighbours are easy to derive from this. Another benefit of using squares is that a given area can be tessellated such that the resulting tessellation maintains an overall rectangular form. This

[9] https://hexnet.org/content/hexagonal-geometry

[10] http://www-cs-students.stanford.edu/~amitp/game-programming/grids/#coordinates

[11] http://strimas.com/spatial/hexagonal-grids/

[12] https://en.wikipedia.org/wiki/Taxicab_geometry

enables the developer to define the borders of the map with four lines, as opposed to the jagged edges that triangle and hexagonal tessellations will cause.

2.1.3.1 N-gon Rendering

In 3D graphics pipelines squares and other quadrilateral polygons do not render correctly when their vertices are not coplanar. Instead the squares are broken down into triangles for rendering. This is the process of triangulating a polygon and it applies to *n-gons*[13]. Hexagons, as a 6-gon, are also triangulated. The effects of this triangulation are visible when viewing the rendered polygon from different angles. This can sometimes be the reason why a hexagon in *Fall* clips through the terrain if the terrain under it is raised. The triangulation can not be controlled easily, but the effects of this are mitigated by elevating the hexes off ground level. See Figure 53 in Appendix I for an example of a situation where this effect is visible. The rendering of meshes is explained in greater detail later in chapter 4.2.1.

2.2 Regular Hexagonal Tessellation

A regular hexagon is a simple polygon with six sides and congruent interior angles. This shape has been featured in board games since as early as the 18th century in a French game called Agon [3]. Regular hexagons can be used as cells in a hexagonally tessellated game map. In video games hexagons provide more options for certain gameplay mechanics. For example when defining the movement logic of a game character on a map where movement takes place whenever a character transitions over the edge of a cell, then using a hexagon instead of a square would provide the character with more edges to transition over, thus providing more directions to move in.

Hexagons are, as in board games, more commonly seen in video games that feature turn-based gameplay. In this scenario hexagons are used systematically in the positioning of *board game pieces*[14] or virtual game objects such that every object in the game is usually associated with one cell at a time (Figure 5)[15]. In other cases larger objects or characters can occupy more than one cell and be associated with several adjacent cells (Figure 4)[16].

[13] https://en.wikipedia.org/wiki/Polygon

[14] https://en.wikipedia.org/wiki/Glossary_of_board_games#pieces

[15] https://uk.gamesplanet.com/game/sid-meier-s-civilization-beyond-earth-steam-key--2612-1

[16] https://steamcommunity.com/sharedfiles/filedetails/?id=970569816

Figure 5. Every unit stands in one particular cell at a time. ***Sid Meier's Civilization: Beyond Earth***.

Figure 4. A larger character occupies more than one cell in ***Blackguards***.

2.3 Coordinate Systems for Hex Maps

There are three primary coordinate systems used for identifying cells in a hex map: offset, axial and the cubic coordinate system. The offset coordinate system has some characteristics that make calculations that involve finding neighbouring cells more complicated and can potentially involve higher computation costs. Axial coordinates are a variant of the offset coordinates, but are still not ideal due to a shortcoming described in the relevant subchapter below. If any cell needs to be able to find its neighbours and the distances to these neighbours as easily as possible, cubic coordinates are the most straightforward option. For this reason *Fall* uses the cubic system. This chapter will give an overview of the cubic system and its alternatives and describe why cubic coordinates were chosen. The author recommends referring to the interactive illustrations presented in the article by RedBlobGames [4] for better visual representations of each system.

2.3.1 Offset

Offset coordinates use two numbers to identify each cell. It is known as the offset coordinate system because alternative rows and columns are offset by half of the width of the hexagon cell while aligning the grid [5]. This convention is very similar to the coordinate system used in square-based maps, where movement across cells either horizontally or vertically will increment or decrement just one of the two arguments. Note that depending on the chosen orientation of the hexes, either vertical or horizontal movement on this hex map does not happen in a straight line, like it does on a square-based map. Instead either horizontal or vertical movement will incur zig-zagging (see Figure 6 where the zig-zagging of the first argument is highlighted in blue). Additionally, the coordinate changes in diagonal movements also vary. Depending on the relative positions of the cells in the overall map, we have to change either just one argument, or both.

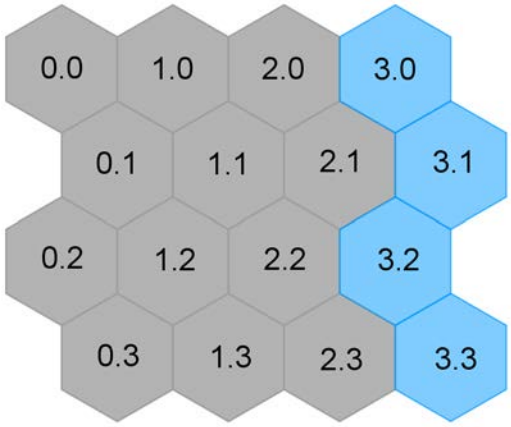

Figure 6. One version of the offset coordinate system where odd rows are offset to the right.

Offset coordinates can be a good choice when knowing the relative position of any cell in the overall map is important. Otherwise offset coordinates are difficult to manage if the calculations performed on hexes are more focused on specific cells and their neighbors. In this regard, axial coordinates may be a better option.

2.3.2 Axial

Axial coordinates are a variant of the offset coordinates. The axial coordinate system is designed to remedy the issues mentioned in the previous subchapter. It improves upon the offset system by recalculating the coordinates of each cell such that incrementing one argument at a time no longer incurs the zig-zagging movement on the map. Figure 7 has highlighted the the sequences of cells (henceforth called an *axis*) that increment or decrement just one argument at a time.

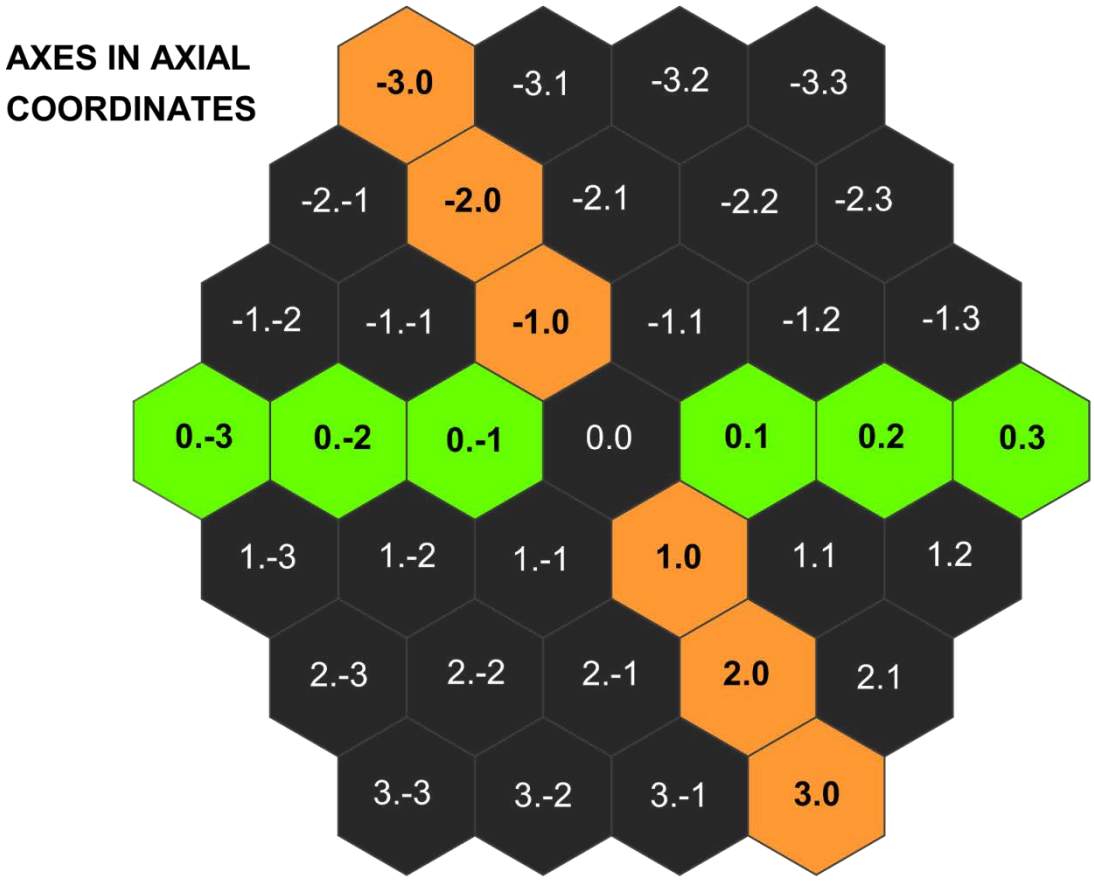

Figure 7. Axial coordinates (axes highlighted)

While making finding neighbouring cells easier than in the offset system, the axial system is still not ideal because the calculations are not uniform across the map: Movement on one axis still requires changing both arguments while movement on the other two axes change just one. This can be fixed by adopting a third argument, which creates the cubic coordinate system.

2.3.3 Cubic

The previously mentioned coordinate systems use two arguments to identify each cell. The offset system had zigzagging and the coordinates of a cell could not be reliably derived from the neighbours without knowing the global position of the cell.

The axial system introduced axes where the coordinate of every cell on this axis could easily be derived from the coordinates of the previous or next cell on the axis, but the problem was that the third axis behaved differently than the other two and required changing both arguments. On Figure 7 this is the axis going through cell 0.0 diagonally from the top right to bottom left (not highlighted).

Cubic coordinates solve the problem with the third axis being calculated differently than the other two by adding a third argument to the coordinates. Now every cell in the map is identified by the arguments $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{z}$ and satisfies the equation $\boldsymbol{x} + \boldsymbol{y} + \boldsymbol{z} = \boldsymbol{0}$. The additional argument has provided us with a coordinate system that has a consistent formula for how the coordinates change when moving along the axes of a cell. Movement in a particular direction always changes the same two arguments: one argument increments and the other decrements. This system is overall more robust and the logic is easier to implement.

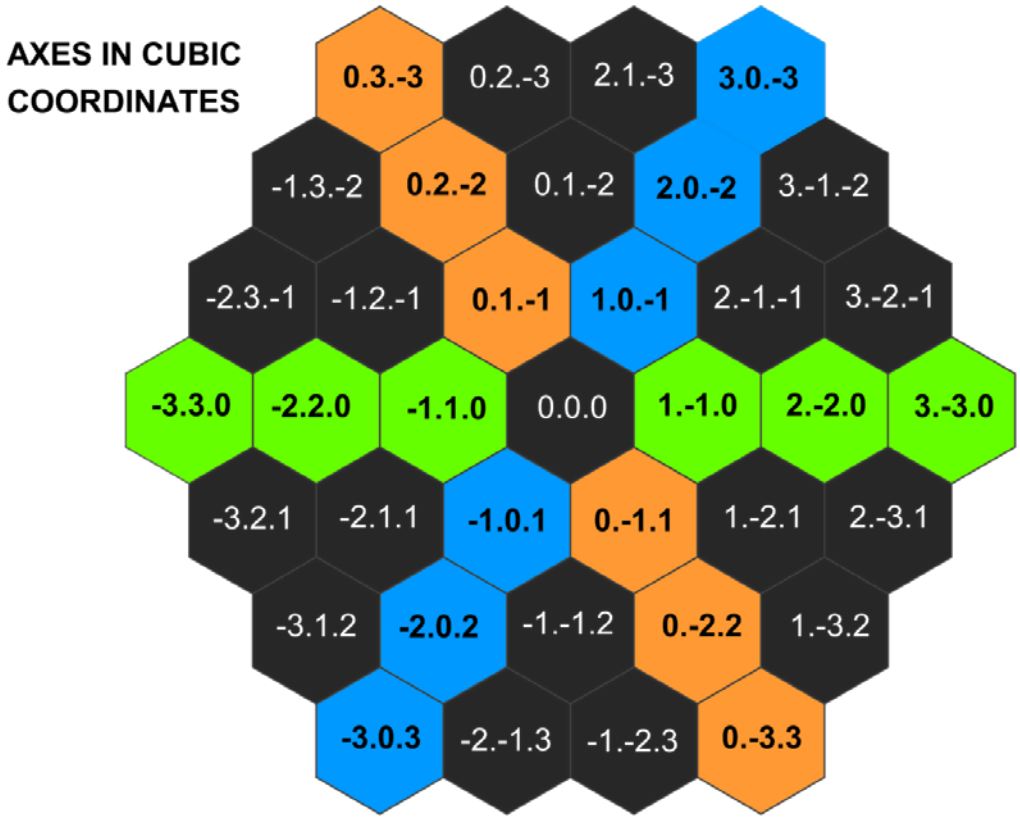

Figure 8. Cubic coordinate system (axes highlighted)

The introduction of a third argument is a heuristic that allows us to easily find all the neighbours of a selected hex simply by knowing the direction of the neighbour we want and changing two of the three arguments. Unlike the offset system, we do not have to know the relative position of the cell. The three axes created by this nomenclature are logically separated based on how the arguments change. See Figure 8, where the hex with the coordinate 0.0.0 is selected and the sequences of cells highlighted in different colours represent the axes around this cell.

This coordinate system greatly simplifies the algorithms for calculating distances, finding immediate neighbours and all the neighbours up to a given distance. These algorithms were implemented for various systems in *Fall* and the specifics of the implementations are described in the implementation chapter, 4.3.5.

The next chapter explores the use of hex maps and other relevant concepts in other games and describes games similar to *Fall* in further detail.

3 Similar Games

This section will start by giving an overview of the different ways hex maps have been used in video games. The author was not able to find the perfect example of a video game that combines all the features attempted in *Fall*, thus the second part will discuss specific games that were found to be the most similar. Testers of *Fall* were also inquired about such a game, but they did not know examples other than ***Heroes of Might and Magic 3***, which is also mentioned in this chapter.

3.1 Hex Maps in Video Games

Hex maps are used in video games in a variety of ways. This chapter explores some of the most common use cases.

3.1.1 Hex Map as Part of a Campaign Map

Not to be confused with a hex map, the term **campaign map** does not necessarily involve any tessellations and instead refers to a small scale representation of an otherwise much larger landscape. The term has been popularized by strategy games that feature empire management gameplay together with another drastically different form of gameplay such as commanding armies in a battle. Classic examples of such games are in the ***Total War*** series, which traditionally has two distinct gameplay modes: **empire management**, which is a turn-based gameplay mode that takes place on the campaign map; **battles**, which is a real-time game mode played in a separate environment that represents an enlarged version of an area on the campaign map. Using the video game ***Shogun 2: Total War*** as an example: on the campaign map two armies could engage in battle near the the city of Kyoto, but to command an army in the battle itself, the player is taken into an environment that is a scaled up representation of the point of conflict on the campaign map. This environment could be a field, forest, desert, city, village or any other place where a historical battle could take place, thus providing a more realistic scale for the battle itself.

Alternatively these battles could take place on the campaign map itself. An example of this is ***Europa Universalis IV*** (Figure 9)[17], which, similarly to *Total War*, uses the character model of a single soldier or its commander to represent the whole army on the campaign map. The animations of this character model and other visuals around it would then represent the battle that in *Total War* takes the player out of the campaign map.

[17] https://www.pcgamer.com/europa-universalis-iv-review/

Figure 9. Two character models, each representing an army battling on the campaign map in ***Europa Universalis IV***

Hex maps are found on the campaign map of some games, most commonly in **4X strategy games**. The term "*4X*" is short for "*explore, expand, exploit, exterminate*" and is a term coined by Alan Emrich in 1993 to describe empire management games that include these four objectives in their core gameplay mechanics [6].

Examples of games featuring a campaign map together with a hex map include ***Sid Meier's Civilization V***[18] (Figure 10)[19], ***Endless Legend***[20] (Figure 11)[21], ***Pandora: First Contact***[22]. In these games the hex map is layered on top of the campaign map and is used to divide the campaign map into distinct segments (cells). In 4X games these cells would be used as containers for various physical game elements, such as characters, buildings, cities, natural objects etc.

[18] https://en.wikipedia.org/wiki/Civilization_V

[19] https://www.instant-gaming.com/en/91-buy-key-steam-civilization-v/

[20] https://en.wikipedia.org/wiki/Endless_Legend

[21] https://anykeytostart.wordpress.com/2015/04/22/endless-legend/

[22] http://pandora.proxy-studios.com/

Figure 11. Various game elements are contained within the cells of the hex map in ***Sid Meier's Civilization V***

Figure 10. The campaign map in ***Endless Legend***

3.1.2 Hex Maps in Arena Combat

In role playing games hex maps are less common and in such games the map is usually reserved for turn-based arena combat scenarios. Some examples of role playing games featuring combat on hex maps are ***Blackguards 1***(Figure 13) and ***2***, ***Heroes of Might And Magic 3***, ***Braveland***. One previously mentioned strategy game also features a hex-based arena: ***Endless Legend***.

The role playing games games listed above, similarly to *Total War*, are separated into two entirely separate game modes. These games also feature a campaign map or some alternative form of gameplay that takes place in a vast environment. For example Figure 12[23] shows what the campaign map looks like in a graphically modernized version of the classic game ***Heroes of Might and Magic III HD***. Figure 13[24] shows the game map in *Blackguards*.

Figure 12. The campaign map in ***Heroes of Might and Magic III HD***

[23] https://www.ubisoft.com/en-gb/game/heroes-of-might-and-magic-3-hd/
[24] https://www.gamewatcher.com/previews/blackguards-2-preview/12104

Figure 13. Campaign maps come in different forms. ***Blackguards*** uses a drawn map which contains various nodes (cities, villages) that the party of the player travels between

These games are distinct in one aspect: they all feature combat scenarios where conflict between the human player and a hostile player (not necessarily artificial intelligence) is resolved in an enclosed space (an environment separate from the campaign map). This enclosed space is the **arena** and it contains the following elements: the **battlefield**, the **characters** controlled by the player or the enemy and a **hex map** on which the characters move and fight.

The size of this hex map is limited by the size of the battlefield. Generally in such games the battlefield is not very large and is significantly smaller than the large map in *Fall*. Using the previously mentioned games as examples, their use of a hex map in an arena setting is pictured on Figure 14[25] and Figure 15[26].

[25] https://store.steampowered.com/app/281770/Blackguards_Untold_Legends/

[26] https://www.gamereactor.eu/images/?textid=277164&id=1339824

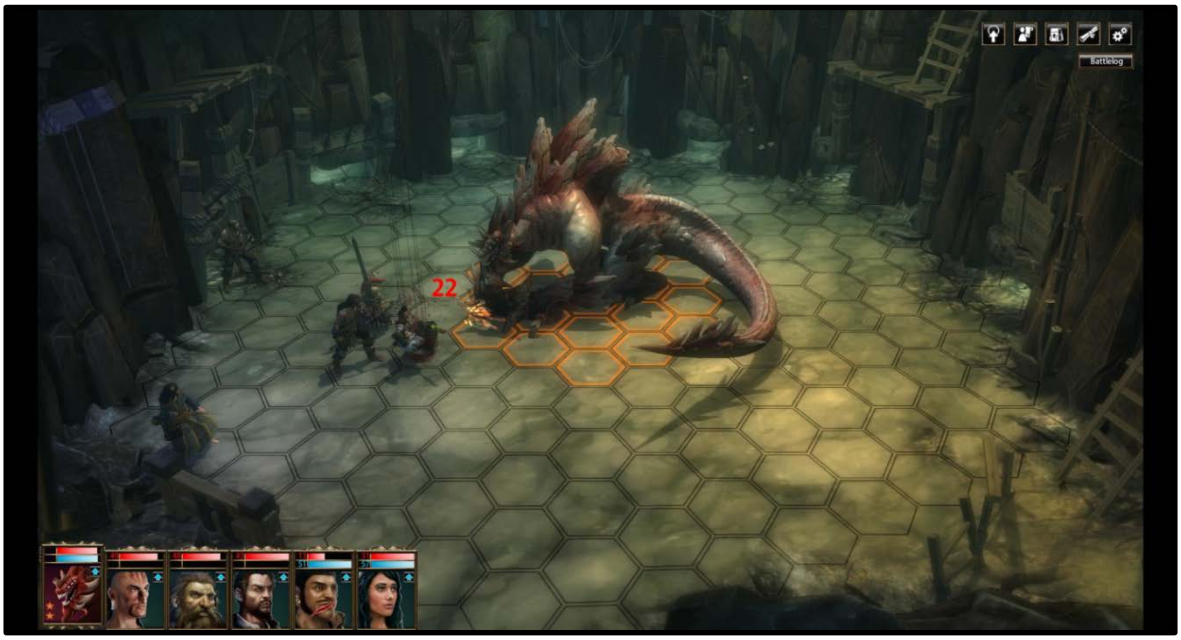

Figure 15. Arena hex map in ***Blackguards: Untold Legends***

Figure 14. Arena hex map in ***Heroes of Might and Magic III HD***

3.1.3 Hex Maps in an RPG Setting

The previous subchapter described role playing games that use hex maps in an arena setting. Combat is a part of most RPG games, but so is exploration. The games described in this chapter combine combat and exploration such that the transitions between combat scenarios and exploration is practically seamless. Combat takes place in a setting similar to the arenas described in the previous subchapter, but in this group of games the arena is merged with the campaign map or simply the game world. ***For the King***[27], ***Fallout 1*** & ***Fallout 2***[28], ***Expeditions: Viking*** and ***Expeditions: Conquistador***[29] are examples of such role playing games that in addition to the aforementioned characteristics feature a hex map in combat situations.

All hex map logic is sometimes disabled and hidden from the player when they are outside of combat; in combat the hex map is enabled. *Expeditions: Viking* is an example of a game that does this. In this game movement outside of combat is freeform: the position of a character is not fixed to the center of a cell at any time and the character can freely move to positions that would in the presence of a hex map be in the corners or on the edges of the cells.

The reason for reserving the hex map for combative situations is simple: movement on a hex map is limiting and the size of the cells in the map define how easy it is for any character to get into the nooks and crannies of the environment. If the cell size is larger, then this type of interaction with the environment is made more difficult. *Expeditions: Viking* involves actions like looting bodies or searching containers. Performing these actions involves a character animation and if the character has to be positioned at the center of a cell, animating this looting action convincingly might require that character to move away from the center of the cell to approach the object being interacted with. After the interaction they could return to the center of the cell so that the concept of characters being associated with particular cells would be visually represented as well.

Aesthetically this is not a good solution. A more aesthetic solution would be what *Expeditions: Viking* has implemented. The game becomes hex-based only when the player engages in combat (Figure 16, Figure 17)[30]. In combat the hex map represents a tactical

---

[27] https://store.steampowered.com/app/527230/For_The_King/

[28] https://en.wikipedia.org/wiki/Fallout_(video_game)

[29] https://store.steampowered.com/app/237430/Expeditions_Conquistador/

[30] https://store.steampowered.com/app/445190/Expeditions_Viking/

playing field where the characters involved are positioned inside the cells of the hex map. The actions performed by the characters are now restricted by their positioning on the map. For more information on how characters are positioned on maps, see chapter 2.2. The use of a hex map provides a certain degree of predictability regarding the actions performed by the characters, at the same time simplifying the planning phases due to the fact that there is a limit to the number of possible actions during a turn. Tactics is a concept that is more relevant in a combat situation and in this game the hex map is a core component in the combat system. *Fall* attempts to extend the use of a hex map beyond these combative situations and instead have it be a part of open world exploration as well.

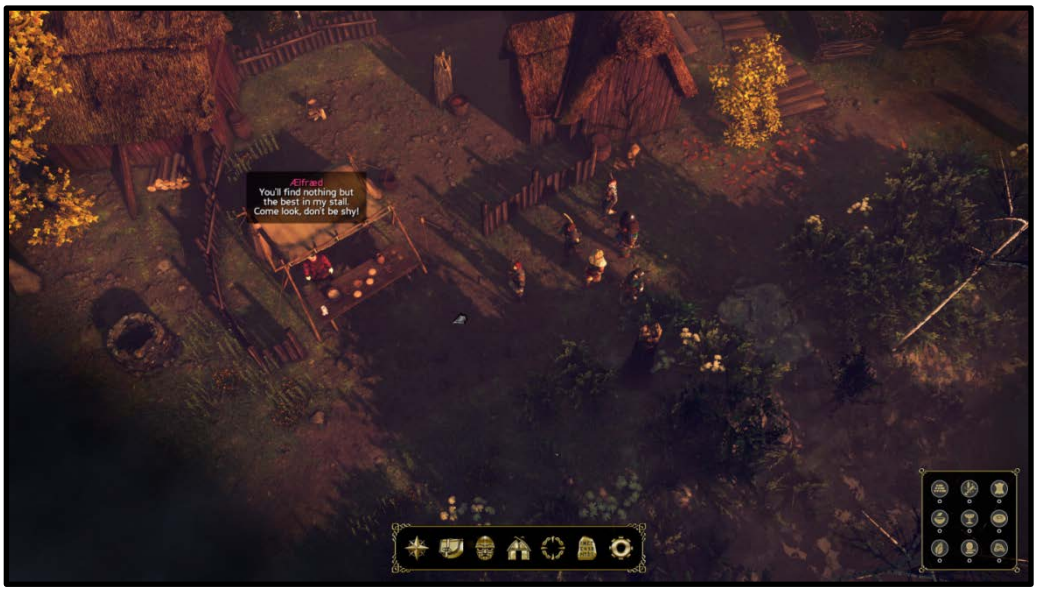

Figure 16. Hex maps are not present when exploring in the party based role playing game

Figure 17. A hex map IS present when engaged in combat in ***Expeditions: Viking***

The cult classics ***Fallout*** (1997) and ***Fallout 2*** (1998) deserve a special mention in the context of this work. *Fall* shares more similarities with these *dimetric*[31] role-playing games than the other games mentioned. Below is a comparison table of some of the features in either game.

Feature	***Fall***	***Fallout***
The hex map overlay is present everywhere: inside and outside of combat	Yes, the map is present everywhere and the layout is clearly visible at all times.	Yes, but is hidden from the player (though internally the characters move on the hex map at all times).
Gameplay becomes turn-based when enemies are nearby	Yes, gameplay becomes turn-based when the player **approaches an enemy**, but combat is initiated when either the player attacks or the enemy sees the player.	Yes, the gameplay becomes turn-based when the player **enters combat** with an enemy (is seen by them)[32].
Seamless open world	The game world is one **open area** where the player can go anywhere.	The game world consists of **separated regions** that the player travels between.
Three-dimensional world with a landscape	Three-dimensional and the landscape is uneven.	Dimetric 2D-graphics on a flat landscape.
Uses an action point system to regulate the availability of different actions per turn	Does not use a point system that is distributed for both types of actions in the game (attacking and moving); instead there is a point system that limits **movement**, but a combative action always ends the turn.	Actions performed by the character consume action points. Action points are used for both **movement and combative** action.
Combat takes place in the same environment as exploration	Yes	Yes, though sometimes the player may be temporarily inside a separate environment(eg. indoors)

[31] https://en.wikipedia.org/wiki/Axonometric_projection#Three_types

[32] https://steamcommunity.com/sharedfiles/filedetails/?id=194805772

3.1.4 Other Hex Map Implementations

Combinations of the aforementioned groups have also been used in games. For instance ***Age of Wonders 3***[33] combines 4X strategy on a campaign map and incorporates arena combat. In this game hex maps are present in both scenarios, but the scale of the world is different in either scenario.

Thea: The Awakening[34] features a hex map solely on the campaign map and does not use it in combat. Combat in this game is essentially a card-based minigame instead.

Most of the games mentioned in this chapter separate exploration gameplay or campaign map gameplay from combat scenarios such that either form of gameplay takes place in separate environments that are functionally independent of each-other. This kind of environmental separation of game modes is useful as a distinction between different game logics. *Fall* attempts to have these two game logics work simultaneously by having the player consider their movements more carefully when exploring in order to achieve a better starting position when they enter combat, which takes place in the same environment. Figure 18 below categorizes the games mentioned.

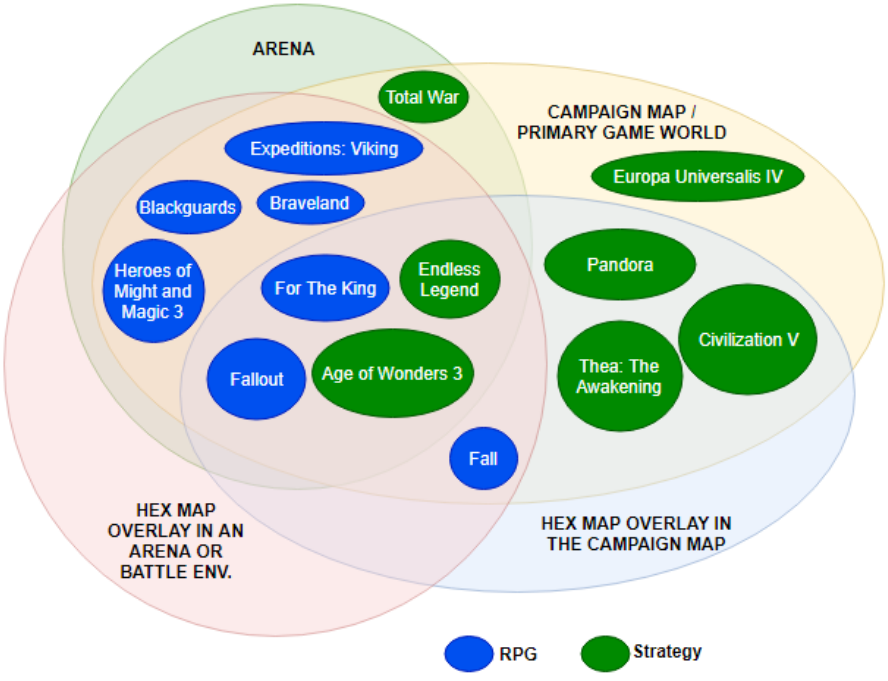

Figure 18. Classification of the previously mentioned games.

[33] https://store.steampowered.com/app/226840/Age_of_Wonders_III/

[34] https://store.steampowered.com/app/378720/Thea_The_Awakening/

4 Implementation

Fall was implemented in the Unity[35] game engine. Most of the models and animations in the game were specially created in the 3D computer graphics software Blender[36] (version 2.79). The models created specifically for this game are described in chapter 4.1.2.1. The game logic itself was implemented in the C# programming language, which is the standard programming language in Unity. Both Unity and Blender were chosen for this task due to the fact that they are free to use and the author has prior experience with the platforms from the courses Computer Games Development and Design (MTAT.03.263) and Computer Graphics (MTAT.03.015).

This chapter will give a comprehensive overview of the systems implemented in *Fall*, describes the existing tools used, custom tools and assets created and details the implementation of the hex map overlay. Due to the large number of systems described in this chapter, some topics will be covered in less detail. Most of the information already provided in the **user manual** is not repeated here, therefore it is recommended to refer to that document (found in Appendix II) first before reading this chapter.

4.1 Tools

This chapter discusses the Unity Terrain[37] tool and how it was used to create the landscape in the game. In addition there is a subchapter about a custom editor tool that was made to decorate the landscape. The second part discusses Blender and will give an overview of the models that were created by the author.

4.1.1 Unity

Unity is regarded as one of the most popular engines for indie games development [7], largely due to its many easy to use features and its compatibility with various target platforms [8]. Unity is consistently releasing new features and patches and developers have easy access to high quality learning materials. Though it has also been criticized for its lack of backwards compatibility even between minor releases [9]. The development of *Fall* was also set back a few times by issues that were found to be either undocumented or unresolved.

[35] https://en.wikipedia.org/wiki/Unity_(game_engine)

[36] https://en.wikipedia.org/wiki/Blender_(software)

[37] https://docs.unity3d.com/Manual/script-Terrain.html

The following two chapters will give an overview of the terrain tool that was used for the creation of the landscape and the custom tool made to replace one of the tools that did not work properly in the Unity terrain editor.

4.1.1.1 Terrain

Unity includes an official tool used to quickly create vast terrains for the game world. This is similar to the Terrain[38] utility in Cryengine or Landscape[39] in Unreal Engine.

Terrain utilities are used to create land which acts as the ground for other systems in the game (without it the game world would be in empty space). It features tools to modify (from now on *sculpt*) a geometry to simulate valleys, mountains, rivers. Terrains are usually calculated from a *heightmap*[40], as opposed to *voxel-based*[41] terrains. Heightmaps are *greyscale*[42] textures where the brightness of a pixel on a heightmap texture directly correlates to the height value of a section on the geometry being generated from it. Textures are usually represented in two dimensions and the flat geometry of a terrain is also rectangular.

Terrain sculpting tools modify the height of the underlying geometry. The Unity Terrain editor provides various tools for creating a detailed terrain. It is possible to sculpt a terrain entirely by hand using special brushes in the Terrain component that is attached to the geometry *gameobject*[43]. It can be a very time consuming task to create large terrains by hand, therefore another method was used to create the underlying geometry of the terrain in *Fall*. Unity can load terrains from existing heightmaps, which are stored in files with the .raw extension. To create a believable terrain, a heightmap based on real world terrain was imported from the online utility at http://terrain.party/. Unfortunately the minimum surface area of the terrain exported from that tool is too large to be imported into *Fall* directly, as it would require a game world that is relative to the real world scale of that terrain. Since the heightmap represents such a large area of land, the texture was edited to suit the needs of a smaller world better by using Adobe Photoshop[44] to modify the *tonality curves*[45],

[38] https://docs.cryengine.com/display/CEMANUAL/Terrain+and+Vegetation

[39] https://docs.unrealengine.com/en-us/Engine/Landscape

[40] https://en.wikipedia.org/wiki/Heightmap

[41] https://en.wikipedia.org/wiki/Voxel

[42] https://en.wikipedia.org/wiki/Grayscale

[43] https://docs.unity3d.com/Manual/class-GameObject.html

[44] https://en.wikipedia.org/wiki/Adobe_Photoshop

[45] https://en.wikipedia.org/wiki/Curve_(tonality)

exposure, brightness and contrast properties on the texture. On the original image most of the landscape is considerably higher than sea level, evident by the overall greyish tone of the image (Figure 22). In real world scale the height fluctuations represented by this texture are spread across a larger area, but in a game where the scope of the world is a fraction of the real scale, the height of the geometry will fluctuate too violently as the vertices are trying to simulate great altitude changes (Figure 19.).

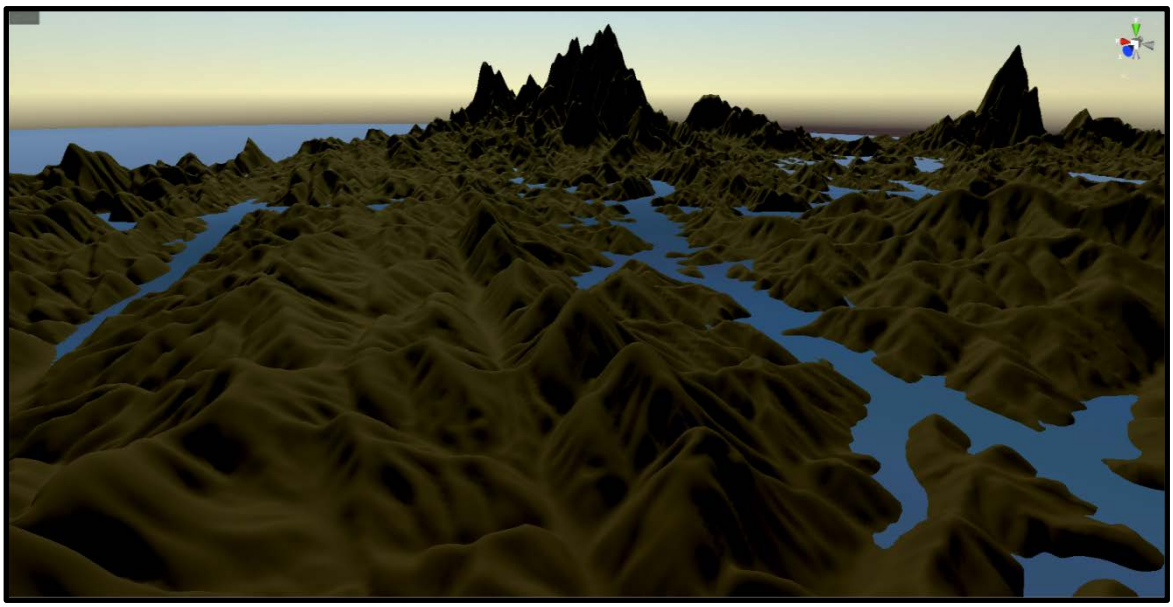

Figure 19. This is what the geometry of the land in Fall would look like if it was just ¼ of the original width and length. This type of compressed world is produced if the heightmap representing a large area is used to generate much smaller terrain

For example, one area that needed to change on the original image is the land around the rivers (Figure 21). The river itself would be visually distinctive even when the geometry is generated from the original heightmap, but the hills that were right next to the rivers were too steep and therefore there was practically no riverbank. While this could simply be remedied by using the sculpting tools to smooth out the bank, the rest of the terrain farther away from the river would still fluctuate too much and make the hexes in the hex map overlay be distorted too often. Therefore the gradient from darker tones to lighter tones was made more gradual across the heightmap, with areas around the river being more dark and areas further away lighter. On the resulting heightmap the hills have less frequent sharp inclines, while increasing the height of the mountain peaks, making them more visible from a distance.

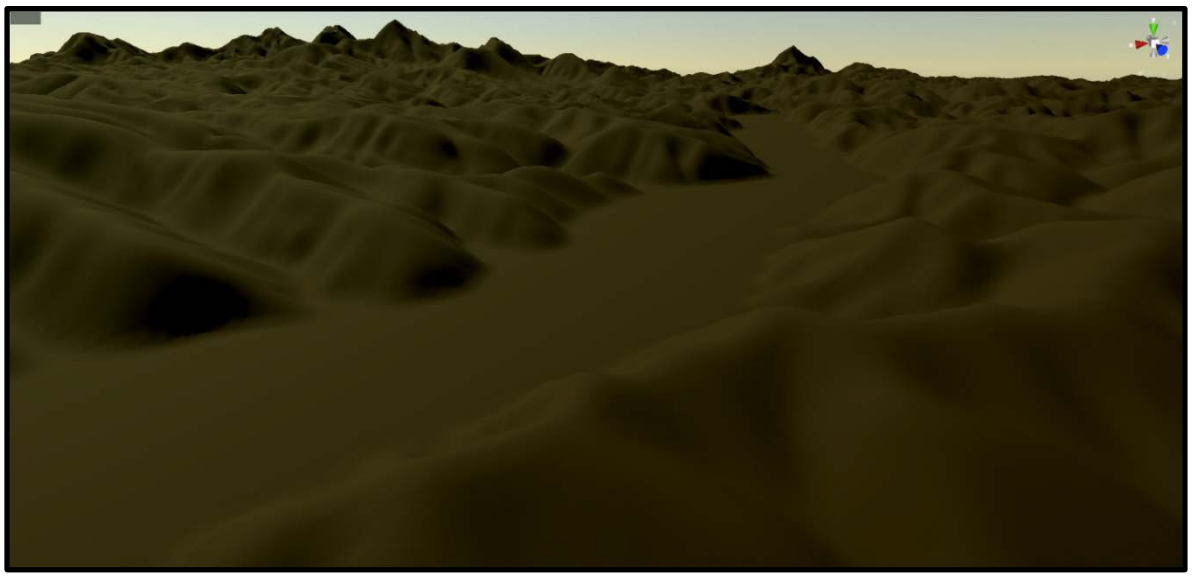

Figure 21. On the geometry generated from the **original** heightmap the hills were too steep, particularly next to the river (the flat area in the middle)

The resulting heightmap (Figure 22) was sufficient for this work as it maintains the key features such as mountain peaks and rivers from the original heightmap and at the same time generates a more stable hilly terrain elsewhere. As a result the hex map will also discard less hexes than it otherwise would. For more information on the discarding of hexes, see chapter 4.3.5. The imported terrain can always be further modified using the brushes provided in the terrain component (Figure 23).

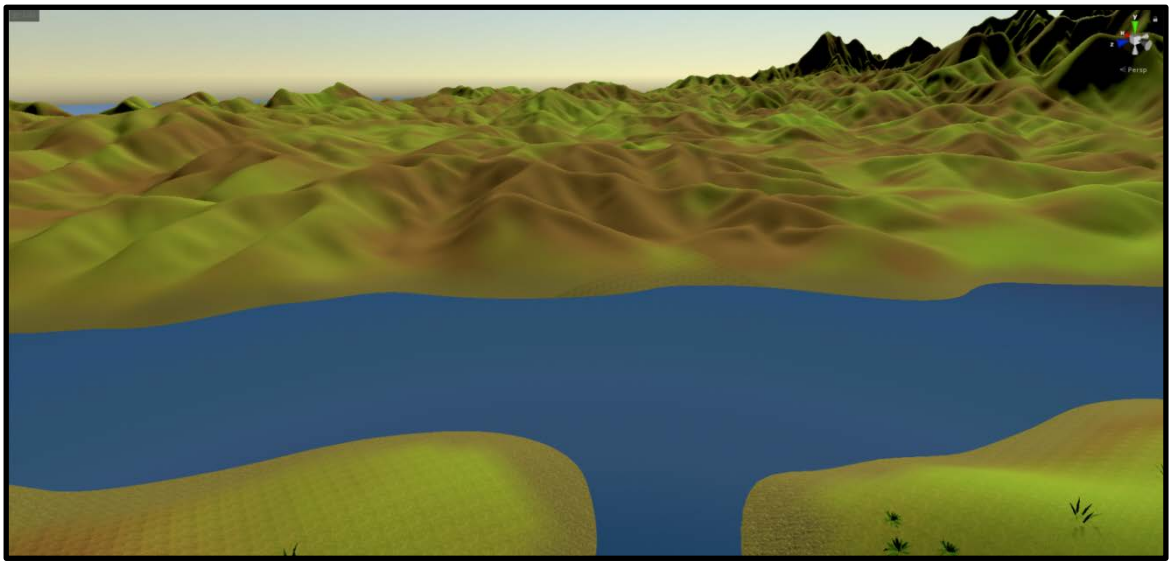

Figure 20. The geometry generated from the filtered heightmap has less steep hills and the overall change in the height of the geometry is more gradual. Note that the riverbanks have been further smoothed by the appropriate brush in the terrain tools.

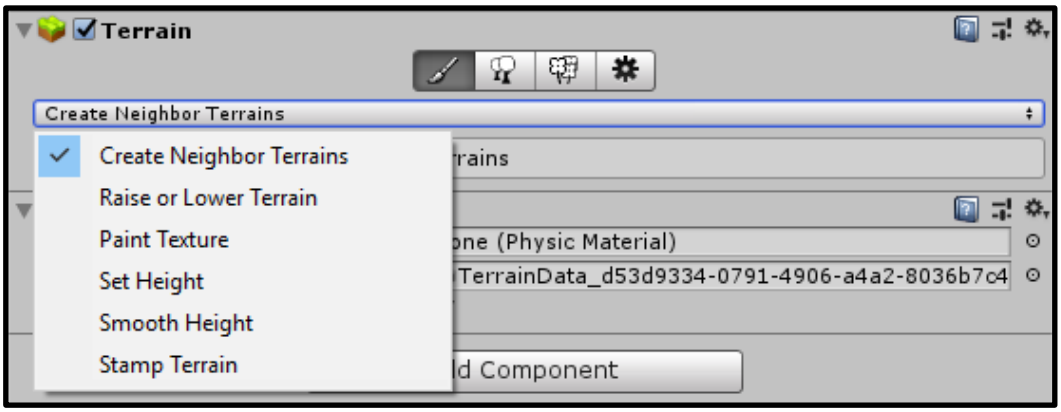

Figure 23. Terrain sculpting tools in Unity 2018.3.9

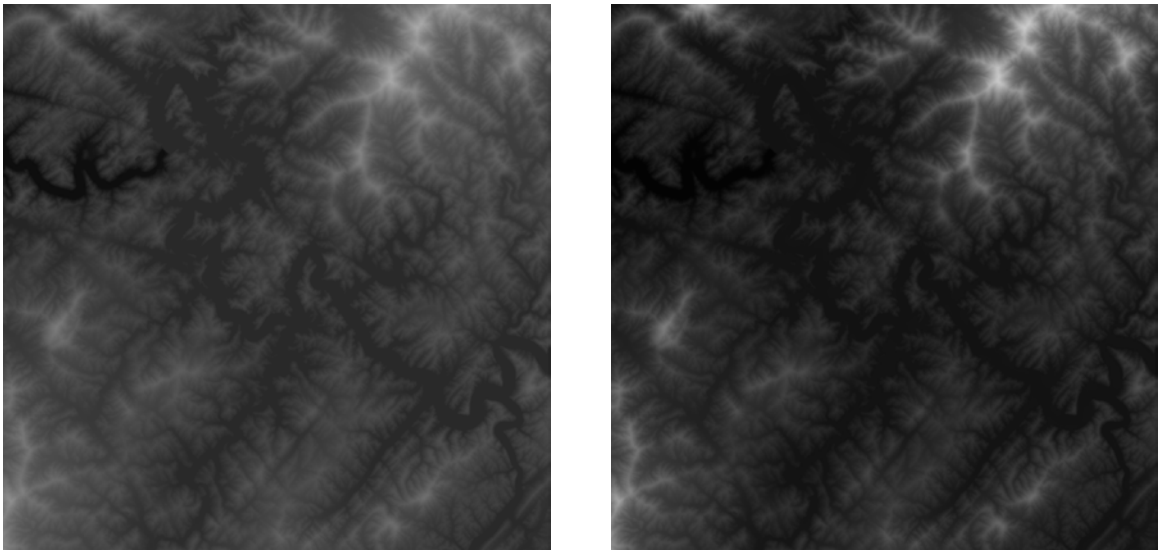

Figure 22. The original heightmap scan and the filtered heightmap

4.1.1.2 Custom Detail Painter

The Detail Painter is a tool bundled with the Terrain component and is used for scattering small decorative objects across the surface of the terrain. Unfortunately this tool failed to work as intended: the tool is supposed to take a *prefab*[46] as input and output instances of that prefab on the terrain, but the tool did not handle these prefabs correctly: they were either not rendered (invisible) on the terrain or their colors were altered in an irreversible way.

For this reason a simple tool for painting details was created as an Unity editor extension (Figure 24). The tool can be opened from the menu path “Window - PaintDetails” on the Unity menu bar.

Currently the tool is hardcoded such the prefabs being painted will have to be placed in the folder “Assets/Resources/1_PREFABS/3_PainterPrefabs”. Any prefab inside this folder

[46] https://docs.unity3d.com/Manual/Prefabs.html

can be selected from the dropdown list and then placed on the terrain simply by clicking on some point on the terrain in the scene view. Unlike the official painter tool, which should paint a large number of the objects on an area the size of the brush, this tool will place objects individually at the cursor location.

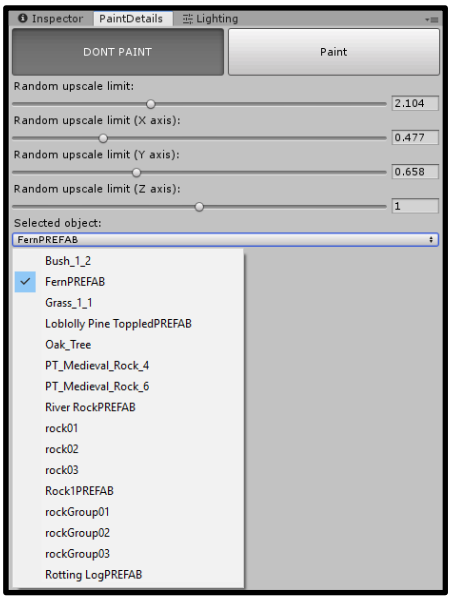

Figure 24. PaintDetails

The tool also takes into consideration the surface normal of the position where the gameobject is placed. The object being placed is rotated such that they always face outwards from the terrain (Figure 25). Objects painted by the tree painter or detail painter in the official toolset are always upright and that was a problem because objects like ferns or rocks would often *clip*[47] through the terrain. In addition, models that did not have a bottom face would make *backface culling*[48] an issue because at some angles the bottom of the model was exposed and the inside of the model was shown as transparent.

Some scaling settings to add variation to the objects are also included.

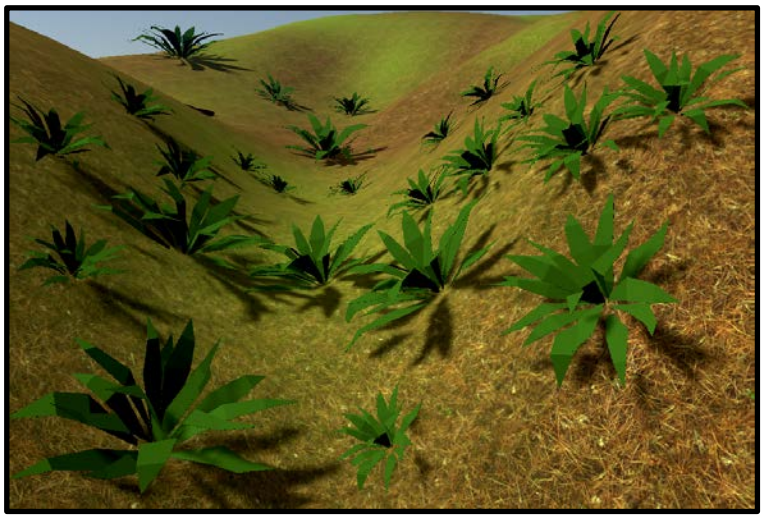
Figure 25. The objects placed are rotated outwards from the terrain

"*Random upscale limit*" defines the maximum multiplier by which the object gets upscaled on all axes. The actual upscale multiplier is a random floating point number between the original prefab scale (by default 1.0) and the number defined by the slider.

The other sliders (Figure 24) work the same way, but the scaling is applied to a particular axis. The number displayed on the right is added to the original scale, which is 1.0. The sum of the original scale and the number on the right is the maximum upscale factor on that axis.

[47] https://en.wikipedia.org/wiki/Clipping_(computer_graphics)

[48] https://en.wikipedia.org/wiki/Back-face_culling

4.1.2 Blender

At the time of writing Blender was in the process of being upgraded to version 2.8, which introduces an overhaul of the user interface and brings various new advanced features. The author acquired most of their knowledge and adopted practises from the Udemy Blender course[49], which was being revised by the instructors to teach version 2.8, but the author chose to continue using the older version (2.79) due to their familiarity with it and considerations regarding potential compatibility issues that the newest version of Blender might have in Unity. The next subchapter gives an overview of the models that were created specifically for this work.

4.1.2.1 Created Models

Several models were created specially for this game. The alternative was to acquire existing assets from either the Unity Asset Store or another 3D asset distribution platform, but the author elected to create a majority of the models manually. The main reason for this was so that the game would have an unique aesthetic that is guaranteed by the models at the least. Having very little prior experience with Blender, this decision led to valuable lessons that gave insight into how the entire process of a model being made in a modelling application eventually ends up being a functional component of a video game.

Every model created for or used in this work is simplistic in appearance and can be considered to be a part of the *low poly*[50] (low polygon count) artstyle. While the term can encompass models that have relatively high triangle counts, it generally refers to models that maintain a low level of detail in their appearance and reuse a small number of *materials*[51] as often as possible. The faces of the models are usually left untextured. Models also commonly use *flat shading*[52], as opposed to *smooth shading*[53]. Smooth shading interpolates the normals of the faces to make a low polygon shape appear more rounded using light reflection calculations. Flat shading maintains the defined outline that smooth shading hides. See Figure 50 in the Appendix for an example of the difference.

[49] https://www.udemy.com/blendertutorial/

[50] https://en.wikipedia.org/wiki/Low_poly

[51] https://en.wikipedia.org/wiki/Materials_system

[52] https://graphics.fandom.com/wiki/Flat_shading

[53] https://en.wikipedia.org/wiki/Shading#Smooth_shading

The **bow** (Figure 26) used by the player is an animated model and it was modelled after a *flatbow*[54].

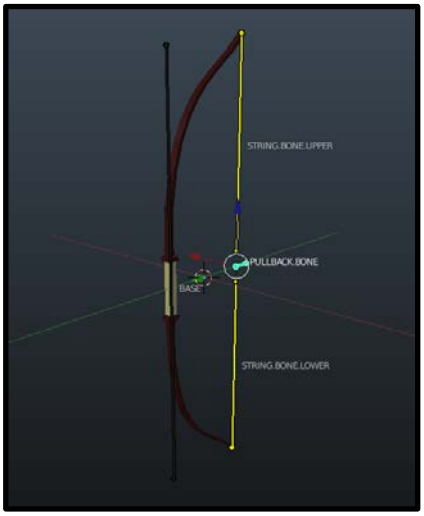

Figure 26. Bow

The **player character** (Figure 27) is modelled as a male human. All parts of the model are moving parts and therefore to make animating more manageable it uses *inverse kinematics* (*IK*)[55] to animate the motion of the arms and legs. IK helps to move jointed parts together in a more natural way by influencing other bones when one bone in the chain is being moved. The player has animations for running, an idle pose, firing the bow, a sneak pose and movement while sneaking.

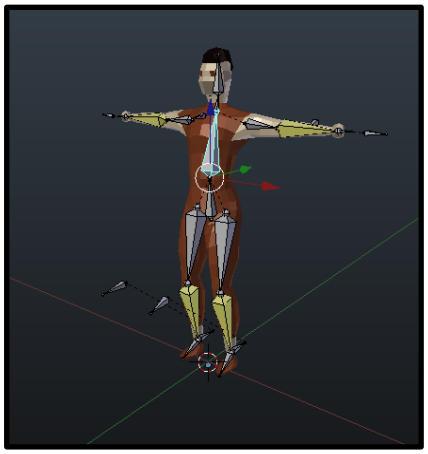
Figure 27. Player character

The **wolf** (Figure 28) is the enemy character in Fall. It was modelled using principles similar to the ones used when modelling the human character, separating the moving parts from the base of the body which runs from the tail along the spine to the head. The wolf is an animated character and contains four animations: idle pose for when they are not alarmed by the player; idle pose for alarmed; running; attacking.

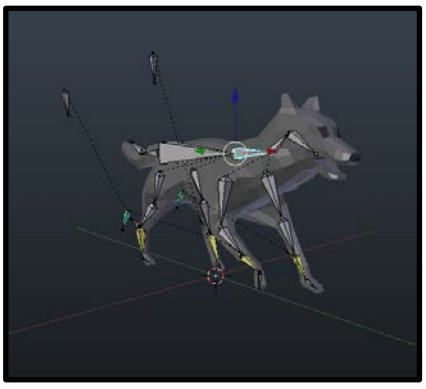
Figure 28. Wolf


Figure 29. Loblolly pine

The loblolly pine (Figure 29) populates most of the terrain. In addition to the base model, the pine has three LOD[56] variants that are needed to reduce the number of vertices rendered. A toppled version of the tree was also modelled.

Other models (Figure 30) created for this work include the fern, red maple, rotting log, extinguished campfire, starting area cliff and a mossy rock.

[54] https://en.wikipedia.org/wiki/Flatbow

[55] https://docs.blender.org/manual/en/latest/rigging/armatures/posing/bone_constraints/inverse_kinematics/introduction.html

[56] https://en.wikipedia.org/wiki/Level_of_detail

Assets not mentioned here were were acquired from the Unity Asset Store or Turbosquid[57]. For an overview of these, see Appendix II .

Figure 30. Other models created

4.2 Hex Map Overlay

The hex map in *Fall* is constructed with a custom painting tool. The tool is used to create neighbouring cells to existing ones or delete existing cells anywhere on the map. This allows the map to be designed in a custom shape instead of a fixed width-height pattern. The custom placement of the cells enables flexibility in the design, particularly around areas that are not suitable for navigation or combat (areas such as rivers, lakes or steep mountains). In addition to restricting movement, custom map shapes can be used as visual indicators that guide the player down a path. One example of this would be a narrow road 3 hexes wide going through a canyon pass that leads to an area of interest. Cells can also be placed on top of other game objects, such as a bridge (Figure 31).

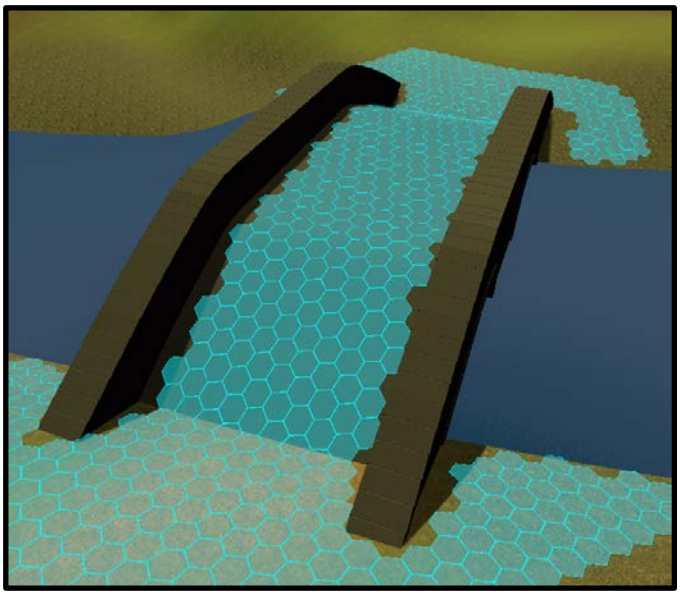

Figure 31. Hexes can be placed on other gameobjects

A hex map generated with fixed width and length parameters can be predictable for the player to navigate. When the player reaches the edge of such a map, they will know that there is no way to cross over to the other side and there is nothing to

[57] https://www.turbosquid.com/

explore in that direction (see Figure 52 in the Appendix).

However, manual design with this tool can result in a map that is more intricate and rewarding for the player to explore. This is particularly useful in larger worlds where reaching the edge of the hex map in one area does not guarantee that there is nothing more to explore beyond that edge.

4.2.1 Rendering

The hex map has to be rendered on top of the surface of the terrain and gameobjects (such as the bridge above). Implementing this was probably the most challenging part because at the same time every hex has to maintain all its functionality and performance should not be impacted to the point where the game is not even playable on any computer.

Two options were considered for rendering the hex map such that it smoothly follows the curvature of the ground. The first option was to use a ***vertex displacement shader***[58]. A vertex displacement shader works at runtime and renders the vertices of a mesh in positions displaced from their origins. The shader does **not** change the original mesh and the effect is merely visual. The displacement of the vertices is calculated by the *graphics processor unit* (*GPU*)[59]. The GPU is performant in rendering calculations and most of the rendering work is delegated to it automatically by the rendering pipeline in Unity itself. *Fall* was built using a beta version of the Lightweight Render Pipeline (LWRP)[60], but the pipeline has not yet been utilized to its fullest potential by customizing the rendering process. Configuring the pipeline to accommodate for the large number of meshes (explained in the next chapter) could be a future optimization that reduces GPU usage.

The *central processing unit* (*CPU*)[61] handles memory management and other processes (such as the operating system) in the background, therefore high memory consumption and over-reliance on the CPU can also slow the game down.

One way that the vertex displacement shader could work in *Fall* is to use the heightmap of the terrain as input for the vertex shader and then have the shader sample the distance by which each vertex has to be moved downwards (the hex map is logically positioned high above the terrain, the reason for this is explained later) using this texture. Provided the terrain scaling settings, the dimensions of the heightmap texture and the mesh of the hex

[58] https://www.jordanstevenstechart.com/vertex-displacement

[59] https://en.wikipedia.org/wiki/Graphics_processing_unit

[60] https://unity.com/lightweight-render-pipeline

[61] https://en.wikipedia.org/wiki/Central_processing_unit

are all taken into consideration, then the hexes would not have to be stored in memory like they are now.

This is not the solution that was used in *Fall* though. The original concern was that using a shader-based approach, the hexes would lose some important functionality. In reality, given a considerably more sophisticated approach this is a future optimization that can help improve performance considerably. Unfortunately it is possible that some features would not work with this approach and other systems would need to be changed drastically. For example the height map that the shader would use does not contain any data about the gameobjects that hexes should be laid on top of and the shader has no direct access to this information, thus complicating the shader-based approach even further.

The hex map is instead implemented by precalculated vertex displacement of the meshes. When a hex is created, the vertices in its mesh are moved downwards and positioned one unit above the surface of the terrain. The target positions for the vertices are found by *raycasting*[62]. Raycasts can selectively filter objects based on their type. The objects in the game are logically distributed into *layers*[63]. These layers can be used to distinguish between surfaces that should support hexes, from the rest of the objects in the game. For example the vertices of hexes should never be in the trees, thus trees are on a separate layer. This enables us to define the surface of the terrain **and** the surface of selected gameobjects as the ground targets for the displacement.

The resulting mesh is stored in an external file and every time the game is first started, the hexes will use their precalculated meshes.

4.2.1.1 Performance Concerns

While the solution described provides the desired result, it also has a considerable performance impact. The original hex prefab has a mesh and every hex gameobject that is instantiated directly from this prefab will have that same mesh. This mesh is shared by all hexes[64]. A thousand hexes could be instantiated, but the memory would contain just one mesh that is referenced by all these hexes.

If the mesh is modified (which is what happens when a vertex is moved around), a new instance of this mesh is created and stored in the memory. This instance contains the same

---

[62] https://en.wikipedia.org/wiki/Ray_casting

[63] https://docs.unity3d.com/Manual/Layers.html

[64] https://docs.unity3d.com/ScriptReference/MeshFilter-mesh.html

amount of information as the original hex and occupies about the same amount of memory.

Performance is noticeably affected when there are high counts of these instances visible in a game world at the same time. Every instance has its own allocated memory space and cumulatively, as the number of instances grow, they consume more memory. Consequently this can also lead to higher latency in CPU and memory communication. In addition, because the hex meshes are different, they can not be *batched*[65] for faster rendering. This could be the main reason why *Fall* experiences performance loss even on high end systems. Some optimizations have been implemented to mitigate this issue. These are described in chapter 5.3.

4.2.2 Map Editing

To understand how new hexes are created, it is necessary to understand what the prefab of a hex consists of.

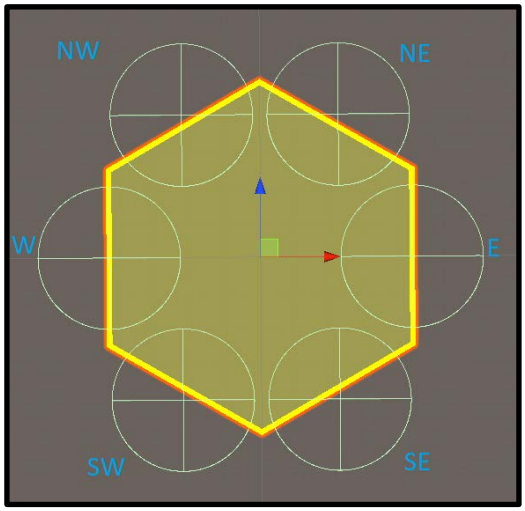

Figure 32. Hex colliders

The prefab of every cell (Hex) has six Sphere Colliders (Figure 32), each one representing an edge of the hexagon. The position of every collider is identified by a compass direction. Since the hexagon used is angled such that the top and bottom are sharp instead of flat, the nomination starts with “NE” (“North-East”) at the top right edge. The colliders are attached to the hex object in the same order.

These colliders are used when generating a neighbour to an existing cell. A neighbour is generated when a left mouse button click or hold is registered on top of any of these colliders. The hit collider will determine the direction in which the new neighbour is generated.

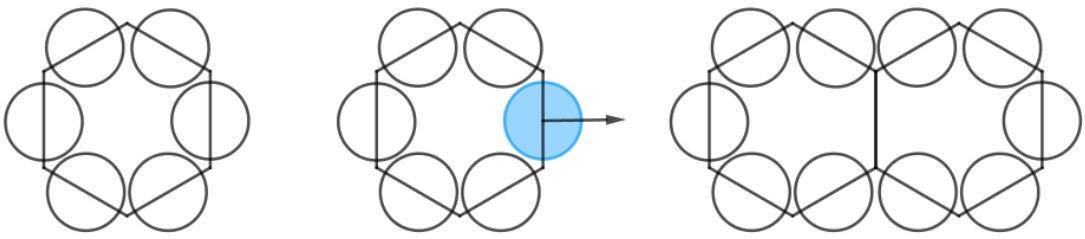

Figure 33. After generating a new cell, overlapping colliders are disabled

[65] https://docs.unity3d.com/Manual/DrawCallBatching.html

When a new neighbour is generated, the colliders of that neighbour and its neighbours have to be re-evaluated to disable all overlapping colliders (Figure 33). A collider is used to expand the grid towards an empty area, but if there is another cell instead, then an active collider would generate a duplicate cell on top of the existing one. Therefore colliders have to be disabled and enabled according to which directions expansion is possible. It is insufficient to disable the colliders on the edge that connects the neighbour with the cell from which it was expanded. A new cell can become a neighbour to several other cells and they all need need to be updated accordingly (Figure 35). The new cell will receive an unique coordinate in the cubic system (cubic coordinates are explained in detail in chapter 2.3.3) and is added to the dictionary data structure (called Map, see chapter 4.3.4) where this coordinate is its unique identifier.

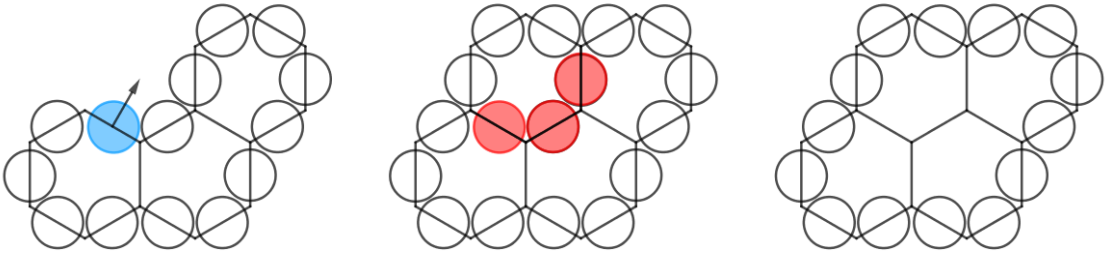

Figure 35. New cells can become neighbours to several other cells

Cells can be deleted by right clicking on a single cell or holding and dragging over them. The cell may have all of its sphere colliders inactive if it has neighbours in all directions, therefore it is not possible to rely only on these to detect mouse input when deleting cells. Since the mesh is hollow in the middle, it will not react to mouse events there by default and the mouse needs another target. One option would be to use a mesh collider that fits perfectly with the shape of the hexagon, but that was not viable because it is inadvisable to use the mesh collider as often as it would be in *Fall*[66].

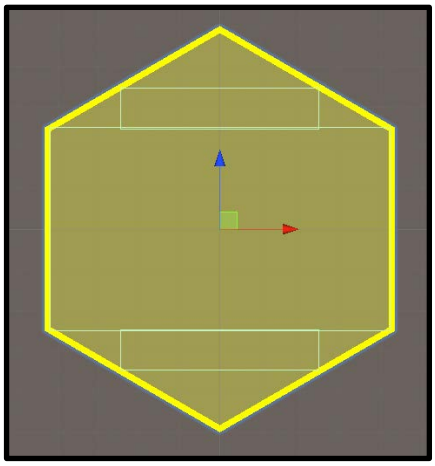

Figure 34. Hex box colliders

An alternative is to use three box colliders instead (Figure 34). These occupy most of the surface area and will fail to register mouse input only when the the mouse is near some of the corners. The colliders do not occupy the entire surface because the colliders can not be rotated. To get the colliders to be at an angle they would instead have to be placed on separate child objects and those

[66] https://docs.unity3d.com/Manual/class-MeshCollider.html

children would be rotated instead[67]. For the purposes here the benefit is very minimal and therefore they were not added. Whenever a cell is deleted, its former neighbours reactivate the colliders used to expand to the position of the deleted cell.

It should be noted that if new hexes are added, the game application needs to be restarted before the added hexes can be used when playing. This is because the editing of the hex map was initially not supposed to be included in the build, but was added later to allow easier access for anyone interested in seeing how it works.

4.2.2.1 Discarded Hexes

The mesh of the hex is always above the terrain and mimics its curvature, but sometimes the newly created mesh can end up looking very distorted if the terrain under it has a sharp incline. Such hexes would appear nearly vertical and they would not be appropriate for any character to stand on. They are identified inside the *HexVertexDisplacer* class, which measures the bounding volume of the newly created mesh. If the difference in the maximum and minimum values of the bounding volume is greater than some threshold (eg. 18.5), the hex is discarded immediately.

4.2.3 Gameplay on the Hex Map

The hex map in *Fall* defines how characters move and fight. This chapter describes some of the systems behind the movement and combat in the game.

Every character is associated with one particular cell at a time at all times. This cell is the *position* of the character. The position is the origin for all actions that can be performed within a certain radius.

All of the following chapters in this section complement existing information provided in the **user manual** (Appendix II), which details the more important specifics of how combat and movement in the game works in different game modes and how movement is affected by the environment.

While some of the logic described here also applies to enemies, enemy behaviour is described in more detail in chapter 4.3.6.

[67] https://answers.unity.com/questions/20779/how-do-you-change-the-orientation-of-a-box-collide.html

4.2.3.1 Movement

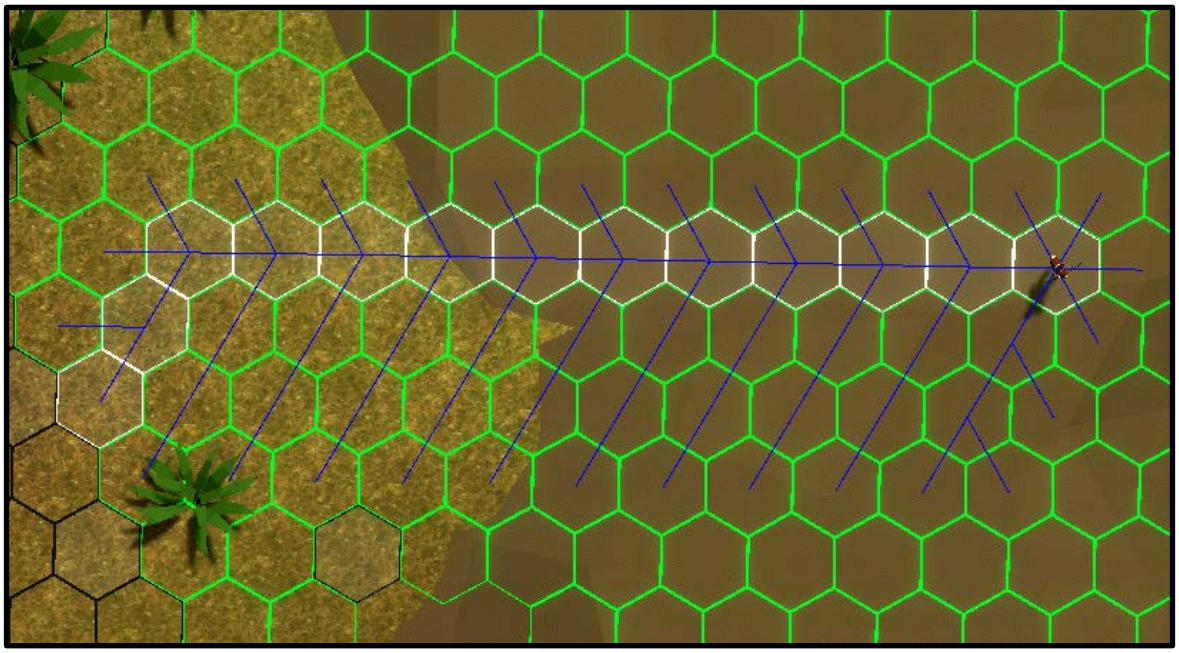

Figure 36. A* pathfinding

All movement is processed inside a class that contains a node graph and the *A* pathfinding*[68] (Figure 36) algorithm that is used to find the shortest path from the position of the character to the selected target cell.

Because the environment in *Fall* contains obstacles that are not associated with specific cell(s), the algorithm needs a way to be made aware of these obstacles and avoid them. This is achieved by using *capsule cast*[69]*:* as A* is searching for the shortest path to the destination, a capsule slightly elevated from the ground travels along the paths being checked and verifies that there are no major obstacles blocking the route there. If there is an obstacle such as a tree, the capsule will collide with it and that particular path will be deemed untraversable. If a path to the target exists and the character has enough movement points to go around the obstacles, the algorithm will return the shortest path avoiding these obstacles.

4.2.3.2 Combat

Combative actions available to the player are also affected by their positioning in the surrounding environment. Please refer to the "Attack Command" section in the manual for a reference of the result.

---

[68] https://en.wikipedia.org/wiki/A*_search_algorithm

[69] https://docs.unity3d.com/ScriptReference/Physics.CapsuleCast.html

As depicted in the manual, a targetable hex has a certain likelihood for the player successfully attacking an enemy standing on that hex. This percentage is determined by capsule casting (Figure 37). Four vertically parallel capsules spaced within 60% of the diameter of the hexagon are sent from the player position to the hexes in attacking range. The path of the capsules mimics line of sight from the head level of the player to the same elevation at the target hex. Every capsule that collides with the environment contributes to the occlusion rating of the target hex and reduces the hit chance there by 25%. If both of the middle capsules collide with the environment, the hit chance is reduced to 0%.

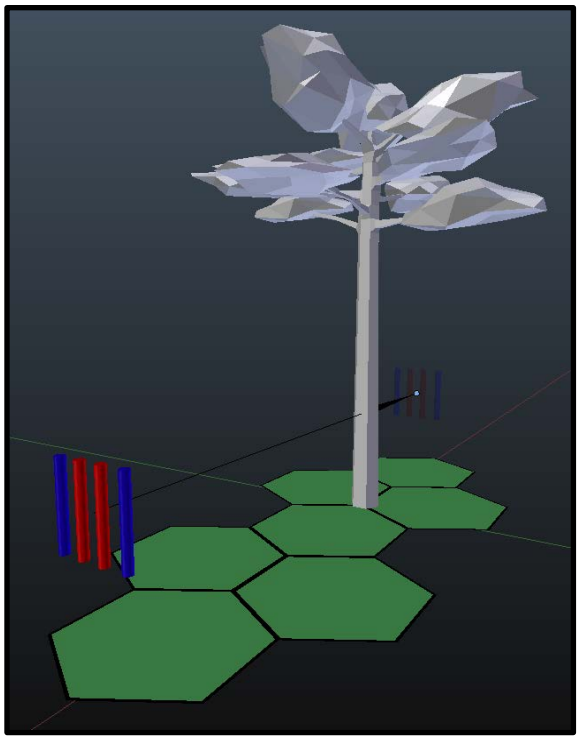

Figure 37. Capsule collisions reduce hit chance

4.3 Architecture

This chapter explains how various systems in the game interact with each-other and how information is communicated between different subsystems. In order to preserve readability, descriptions will be kept relatively broad and will not explain every part of every system thoroughly. Instead the chapters will provide a general overview of the purpose of each subsystem and how they interact with each-other. For further information, please refer to the code base attached in Appendix II.

4.3.1 Game Control

Game Control is a *service locator*[70] *singleton*[71] and it is associated with one gameobject. It contains and manages data and functionality that needs to be uniform across the project (static) or just easily accessible by several classes. Some of its key functionality is described below.

[70] https://en.wikipedia.org/wiki/Service_locator_pattern

[71] http://gameprogrammingpatterns.com/singleton.html

4.3.1.1 Player State

During gameplay the player is always in one of two game modes: exploration mode or combat mode. These are also detailed in the user manual. Player states (Figure 38) are related to this, but are a slightly extended concept. States keep track of what the player is currently doing and lets other components know to act accordingly. The green bubbles represent actions that the player can make. Everything else describes the processes around the player actions. The diagram assumes that whenever the player is issuing a move command they have at least one movement point remaining and attack commands always have a target.

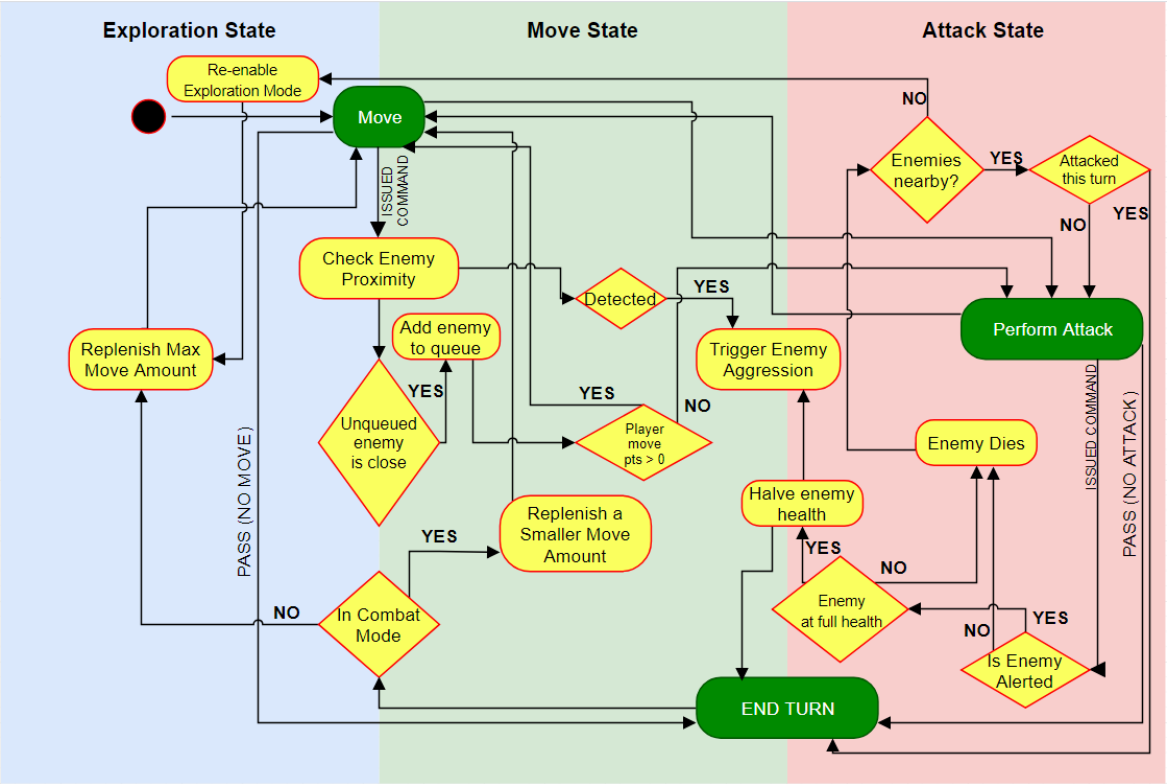

Figure 38. Player states

4.3.1.2 Node Graph

Game Control contains the node *graph*[72] used by A* pathfinding. For the purposes of separating hex logic from graph logic, every Hex is converted into a Node. The graph is supplied with a *dictionary*[73] that connects every hex to their node equivalent. All nodes are initialized and the dictionary constructed when Game Control is created. The graph is then

[72] https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)

[73] https://en.wikibooks.org/wiki/A-level_Computing/AQA/Paper_1/Fundamentals_of_data_structures/Dictionaries

stored in Game Control for the movement system to reference. The main purpose of this *decorator*[74] object is to return the shortest viable path between two hexes.

4.3.1.3 Other Functionality

Game Control contains the method ***InitilizeBaseGameState*** that is run when the game is started. Not to be confused with the player states mentioned earlier, the initial state referenced here is essentially the process of making sure that every time the game is started, every component that might have been modified during gameplay is exactly the same as it originally was. While the game would initially load into its base state without the help of this function, it is needed when an already running gameplay session is restarted via the ingame menu.

As mentioned in the introduction to this subchapter, Game Control contains static data. This data includes **references to static variables and core systems** such as the hex map, node graph, mouse managers, canvas (user interface), turn controller, enemies and the player. Systems that need access to this data need to do it through Game Control. This helps guarantee a more linear processing flow and data is less likely to be accidentally duplicated or modified concurrently by different systems.

The hex map is stored inside an external data file and Game Control initiates the **file input-output** operations to read or write data from or to this file, but the process itself takes place inside the Map class.

The task of populating the world with enemies and relaying their awareness of the player to the user interface is delegated to Game Control. The proximity of the player to the enemies is also being coordinated here. That includes tasks such as adding nearby enemies to the existing turn queue or first starting turn-based gameplay if the player was not in combat mode before.

4.3.2 Turn-based Gameplay

Turn-based gameplay becomes active when the player enters combat mode (Figure 39). Actors are characters that perform actions on their turn. Turn-based gameplay in *Fall* is managed by the *mediator*[75] class Turn Controller, which keeps track of all the Actors and implements the following functionality:

[74] https://en.wikipedia.org/wiki/Decorator_pattern

[75] https://en.wikipedia.org/wiki/Mediator_pattern

- Defining a queue that describes the order in which the Actors perform turns;
- Notifying the next Actor when their turn has started;
- Modifying the turn queue in reaction to changes on the playing field;
- Requesting updates from other classes to have them reflect the start of a new turn

The next chapter will explain the behaviour of the turn controller in more detail.

4.3.2.1 Turn Controller

Turns are managed by Turn Controller, which contains a queue of Actors that are located close enough to the player character for their actions to be of relevance to the decisions of the player. If a character is too far from the player, then they are excluded from the turn queue (are not Actors) and do not move around. When a character is at a fixed distance from the player, that character is added to current queue (becomes an Actor) and is removed from it when the character dies. It is assumed that the player has the intention of killing any enemies they find since that is the objective of the game. Therefore there is no condition that removes characters from the queue when they end up being farther away from the player.

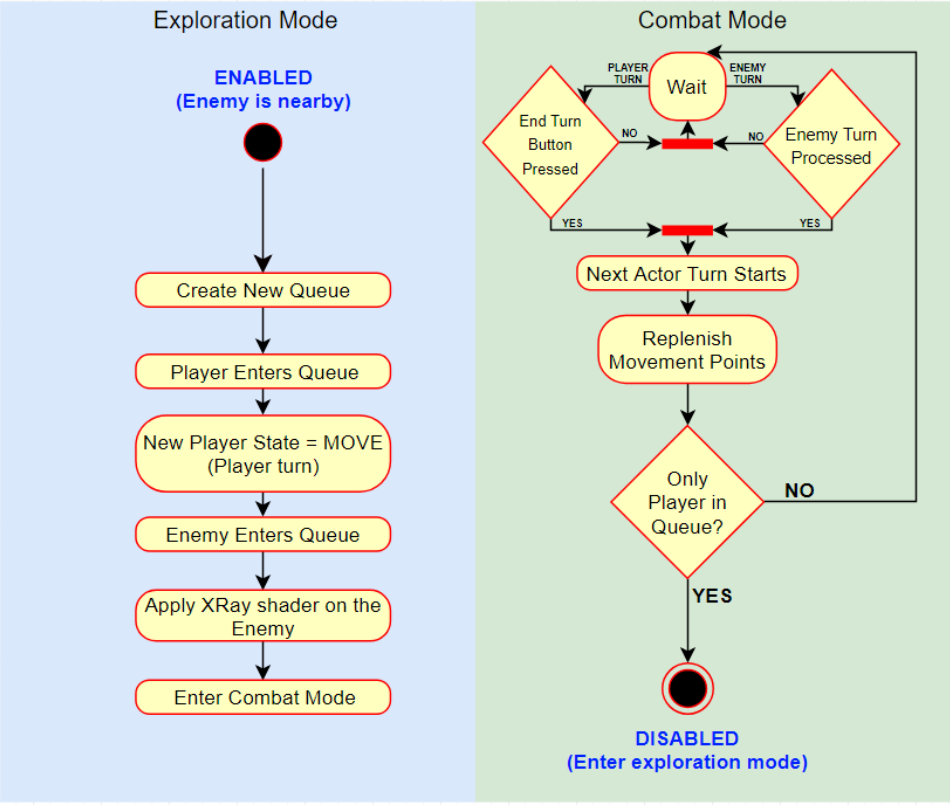

Figure 39. Exploration and combat mode

At the beginning of each turn, the head of the queue is removed, processed and reinserted into the queue.

4.3.3 Mouse Manager

Mouse input is parsed by mouse managers. There are two mouse managers and only one can be active at a time. One is active during gameplay, the other is used for editing the hex map. Both are attached to the Game Control gameobject and are enabled or disabled accordingly. The manager is where the main update loop runs. This chapter will give a rundown of what either manager does.

4.3.3.1 Game

The game mode mouse manager behaves slightly differently depending on whether the player is currently exploring or is engaged in combat. When the player is exploring, all input is processed such that only movement actions are possible (the player can still preview the attack mode highlighting). The manager processes input in combat mode when the Turn Controller is enabled (the controller is enabled only when in combat mode). The manager checks the on/off state of the Turn Controller instead of the player state because the player is allowed to use the attack command button outside of combat in exploration mode to see the highlights on surrounding hexes, but is not in combat.

Mouse input is registered on hexes, but also reacts to mouse clicks on the enemy. The manager is disabled during the enemy turn or when the player is moving. Therefore when the manager is enabled at the start of the player turn, it calls to Game Control to check for new enemies to add to the queue.

4.3.3.2 Editing

The editing mouse manager handles input for left (from now on *LMB*) and right mouse button (*RMB*) clicks or holding. LMB is used when creating hexes and RMB deletes hexes. The logic of how hexes are created was described in a previous chapter, 4.2.2. The implementation itself resides mostly inside the mouse manager.

The editing mouse manager registers input on hexes. When LMB hits any of the sphere colliders attached to a hex, it determines which sphere collider it hit and uses that information to determine the direction in which to generate the new hex.

While the meshes of the hexes follow the curvature of the ground, the colliders used in expanding the map are positioned higher and are coplanar. Their positioning is illustrated in chapter 4.3.8.1. Knowing that the colliders are all planar (the *y*-coordinate, which is the 'up' axis in Unity, is the same for all hexes), we can construct the following *shift vectors*:

```
// Shift vectors
Vector3 northeast = new Vector3(hexHalfWidth, 0, 1.5f * hexHalfHeight);
Vector3 east = new Vector3(hexWidth, 0, 0);
Vector3 southeast = new Vector3(hexHalfWidth, 0, -1.5f * hexHalfHeight);
Vector3 southwest = new Vector3(-hexHalfWidth, 0, -1.5f * hexHalfHeight);
Vector3 west = new Vector3(-hexWidth, 0, 0);
Vector3 northwest = new Vector3(-hexHalfWidth, 0, 1.5f * hexHalfHeight);
```

These vectors are used to position the newly created hex correctly. The end position is the shift vector added to the position of the hex from which the new hex was expanded.

4.3.4 Map

Map is a class that contains references to all the hexes. This data is contained in a dictionary with the same name. This class contains methods for getting access to a specific hex using the unique identifier (coordinate) of that hex; serializing the map for storage in an external file; de-serializing the map acquired from an external file and loading it into the game world; getting a list of random hexes (eg. for use by the enemy, whose behaviour is described in chapter 4.3.6).

4.3.5 Hex

This subchapter describes the functionality that each hex has. Functionality related to the editing of the hex map will not be covered here. For information on how hexes are created to form the map, see chapter 4.2.2 above.

Being the focus of this thesis, Hex is the most versatile class. Many of the systems described elsewhere in chapter 4 call to functions provided here and the purpose of these functions is directly correlated with those systems. Therefore such functions will not be specifically addressed here as they are already integrated into the descriptions of the appropriate systems. For example the Hex class contains the function that enables and disables colliders used in map editing, but that logic is described in another chapter.

Instead, this chapter will provide insight into two hex algorithms that have not been explained elsewhere.

4.3.5.1 Finding Immediate Neighbours

Earlier in chapter 2.3.3 it was mentioned that cubic coordinates simplify the process of finding the neighbours of any hex. Now that we have a dictionary (Map) that contains references to all the hexes, we can use it to find any hex using its coordinate. If we have a hex cell at the coordinate (x, y, z) and we want to find its immediate neighbours, we first have to determine the coordinates that the neighbours have. The cubic system makes this really easy because we already know what the coordinates of the neighbours would be (if

they exist) without having to determine the global position of the cell first. The IDs of the neighbours are constructed as follows:

```
string neighbour_id_NE = (x-1) + "_" + (y+1) + "_" + (z);
string neighbour_id_E = (x) + "_" + (y+1) + "_" + (z+1);
string neighbour_id_SE = (x+1) + "_" + (y) + "_" + (z+1);
string neighbour_id_SW = (x+1) + "_" + (y-1) + "_" + (z);
string neighbour_id_W = (x) + "_" + (y-1) + "_" + (z-1);
string neighbour_id_NW = (x-1) + "_" + (y) + "_" + (z-1);
```

Alternatively the string-based ID-system could yield slightly better performance if re-worked into a 3-tier dictionary instead. Regardless, the current solution has not incurred noticeable performance losses and works fine. Once the coordinates have been constructed the next step is to ask Map if a hex with such a coordinate exists. If it does, it is added to the list of immediate neighbours.

4.3.5.2 Finding Distant Neighbours

The algorithm for finding distant neighbours is very similar to finding the immediate neighbours. The algorithm is again simplified by the use of cubic coordinates. Cubic coordinates guarantee that given a distance *d* and some hex *A* with the coordinates (*x*, *y*, *z*), every consecutive hex on an axis of hex *A* increments one coordinate argument by 1 and decrements another by 1 to satisfy the expression described in the relevant chapter, 2.3.3. This means that the neighbouring hexes do not have an argument that differs from an argument in the coordinates of cell *A* by more than the provided distance *d*. Knowing this fact we can find all the neighbours up to a distance using the following algorithm:

```
public List<Hex> GetDistantNeighbours(int distance)
{
    List<Hex> neighbours = new List<Hex>();
    for (int dx = -distance; dx <= distance; dx++)
    {
        string id_x = (x + dx).ToString();
        for (int dy = -distance; dy <= distance; dy++)
        {
            string id_y = (y + dy).ToString();
            for (int dz = -distance; dz <= distance; dz++)
            {
                string id_z = (z + dz).ToString();
                string id = string.Join("_", new string[] {id_x, id_y, id_z});
                if (GameControl.map.HexExists(id))
                {
                    Hex hex = GameControl.map.GetHex(id);
                    if (!hex.blocked) neighbours.Add(hex);
                }
            }
        }
    }
    return neighbours;
}
```

However this algorithm may not be the most optimal solution, as it will generate keys for hexes that can never exist in the map (such as 3;-3;-3). Regardless, no considerable performance loss from this function has been observed when used by the highlighting method *FilterAttackableHexes*, which is called every time the player starts issuing an attack command.

4.3.6 Characters

Character is an abstract class that manages behaviour shared by the player character and the enemy (wolves). The Player and Enemy classes derive from this class.

The Character class contains a method for moving the character to a new position (hex). The process is roughly the same regardless of whether the character is the player character or an enemy. When a character first starts moving, mouse input and the UI buttons are disabled for the duration of the motion. The method then requests a path from the node graph and supplies that path to MoveCoroutine, which ensures that the characters move along the provided path until they reach their destination.

The Player class contains the function for enabling and disabling sneaking. The function also makes sure that no movement points are lost when going in or out of sneaking mode.

Enemy and Player both have attacking functionality and unlike movement, this is different for either type of character. The attacking functionality on the player requires a weapon (the bow) and the target has to be in attacking range of the weapon. The wolf on the other hand is always a close quarters combatant and as such always has an attacking range of 1, which means that they can attack the player character only when they are positioned on a neighbouring hex. The player on the other hand can use the bow to attack over a greater distance.

The artificial intelligence of the wolves is very basic, particularly regarding their decisions when moving around the map. Some testing sessions made the tester feel like the wolves were always running away from them, thus encouraging the tester to abandon the stealthy approach in favour of a direct charge. If on the next turn the wolves decided to move back in the general direction of the player character, they would often see them standing in the open and thus become alerted, making the wolves less vulnerable to damage when attacked and more likely to hurt the player on the next turn.

In reality even when it might seem like the wolves are intelligent, they are actually not programmed to use any cues when determining a path. Instead they simply select a random nearby hex as the destination.

The wolves become alerted (triggered) when the player character is at a certain distance from them. This trigger distance is reduced when the player is sneaking. Unlike some other systems in the game, the radius of the hexes that trigger aggression by the wolf is not dependent on the environment. Therefore the wolves can be triggered even when they do not have direct line of sight to the player. The Player class has a function that lets the player know which hexes will trigger a wolf when they are issuing a move command (refer to the instructions sheet for a visual representation of this).

The wolves make sure that they do not position themselves on occupied hexes. If a hex next to the player is already occupied by a wolf (or is blocked by the environment), then other attacking wolves will instead try to find another unoccupied hex from the vicinity of the player as their move target.

4.3.6.1 Enemy X-Ray Shader

Nearby enemies become visible through the terrain and are highlighted by a red hologram outline when the player is in combat mode (Figure 40). This effect is created by shading the materials on the wolf with a shader called XRay. The shader was implemented by combining the elements presented in the shader article by Linden Reid [10] with the template LWRP shader[76] (By phi-lira on Github) and adopting the hologram *pass*[77] from the X-Ray shader authored by Jens-Stefan Mikson[78].

Figure 40. In combat mode the wolves can be seen through the terrain

Whenever fragments on the wolf are not occluded by anything, they are rendered with the standard shader pass that simply colors the fragments grey. Whenever the fragments are occluded, they are rendered with the hologram pass. The hologram pass is always run after

[76] https://gist.github.com/phi-lira/225cd7c5e8545be602dca4eb5ed111ba

[77] https://docs.unity3d.com/Manual/SL-Pass.html

[78] https://github.com/mikson60/VisualEffects/blob/master/CG_VFX/Assets/Shaders/XRay.shader

all the other passes because it does not write to the *z-buffer*[79] and its *ZTest*[80] is set to "Greater". Stencil buffer is used to determine when the hologram pass should **not** color the fragment and instead keep the fragment as is (after being drawn by the standard pass).

4.3.7 Bow and Arrow

The Bow script is responsible for spawning arrows and determining whether they will miss the target or not. If the shot is deemed to miss, the arrow will be slightly diverted from its otherwise direct course to the target. To ensure that the arrow does not impact the target by accident, its *collision matrix*[81] is temporarily changed such that the arrow is allowed to collide only with the environment.

The Arrow script is attached to every arrow and it determines the amount of damage dealt upon impact with an enemy. The base damage amount is defined in a *scriptable object*[82] that contains the weapon stats for the bow. This object also contains the damage bonus modifier applied when the enemy hit by the arrow was not alerted.

4.3.8 Camera

There are two types of cameras used in the game: orthographic and perspective. The orthographic camera is used when editing the hex map; perspective camera is used during gameplay. This chapter describes these two cameras and explains why the orthographic camera was preferred for its purpose.

4.3.8.1 Orthographic

The hex map on ground level can be very uneven and therefore it would be difficult to add or remove hexes efficiently when viewing the hexes at an angle. For this reason the collider system described in chapter 4.2.2 is positioned high above the terrain and is interacted with through the "lens" of an orthographic camera[83] that looks down at the hex map. See Figure 41 below.

The book Fundamentals of Computer Graphics [11, p. 71] defines parallel projection as a type of projection in which points in three-dimensional space are mapped on a two-dimensional image by moving them along a projection direction until they reach the im-

[79] https://en.wikipedia.org/wiki/Z-buffering

[80] https://docs.unity3d.com/Manual/SL-CullAndDepth.html

[81] https://docs.unity3d.com/Manual/LayerBasedCollision.html

[82] https://docs.unity3d.com/Manual/class-ScriptableObject.html

[83] https://en.wikipedia.org/wiki/Orthographic_projection

age / projection plane (Figure 42). If the image plane is perpendicular to the view direction, the projection is orthographic. This is the image seen by the orthographic camera.

The hex map, which on ground level mimics the curvature of the terrain, appears to the orthographic camera like a regular tessellation of hexagons. In this view, the hexes look identical (on ground they may be distorted).

Since the colliders involved in editing are above the ground, there is less clutter on ground level, essentially negating the risk of editing tools interfering with game scripts. All the aforementioned benefits make adding and deleting cells considerably easier than it otherwise would be if perspective camera was used instead.

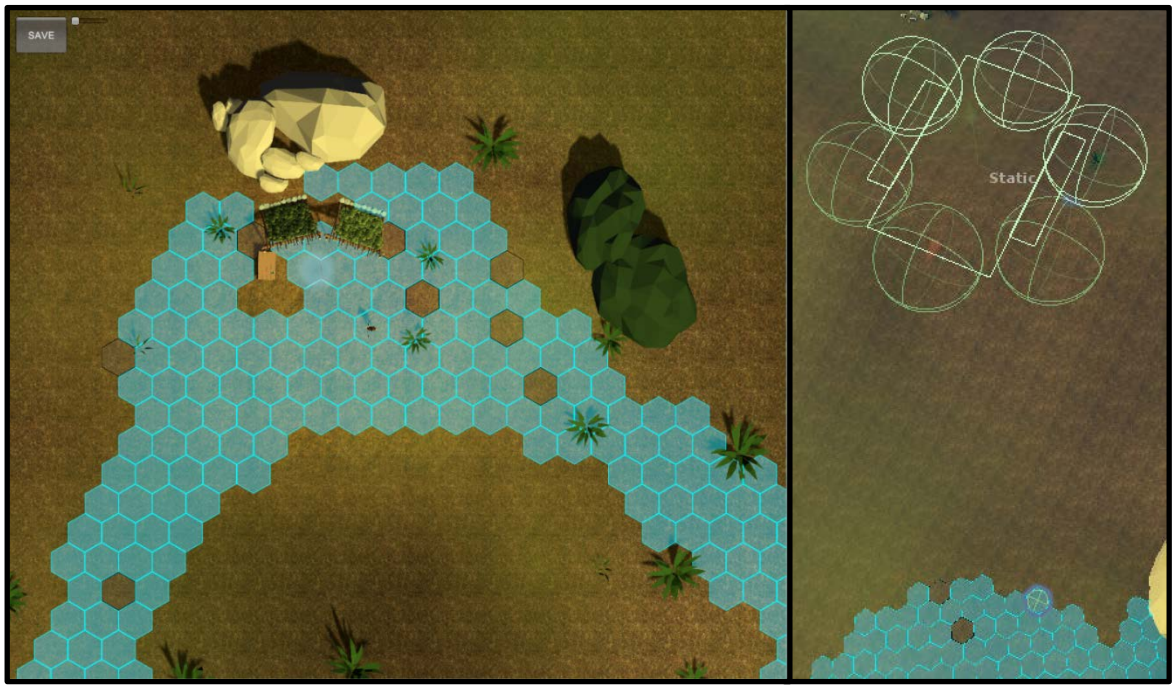

Figure 41. View from the orthographic camera (left). Colliders are elevated from the ground (right)

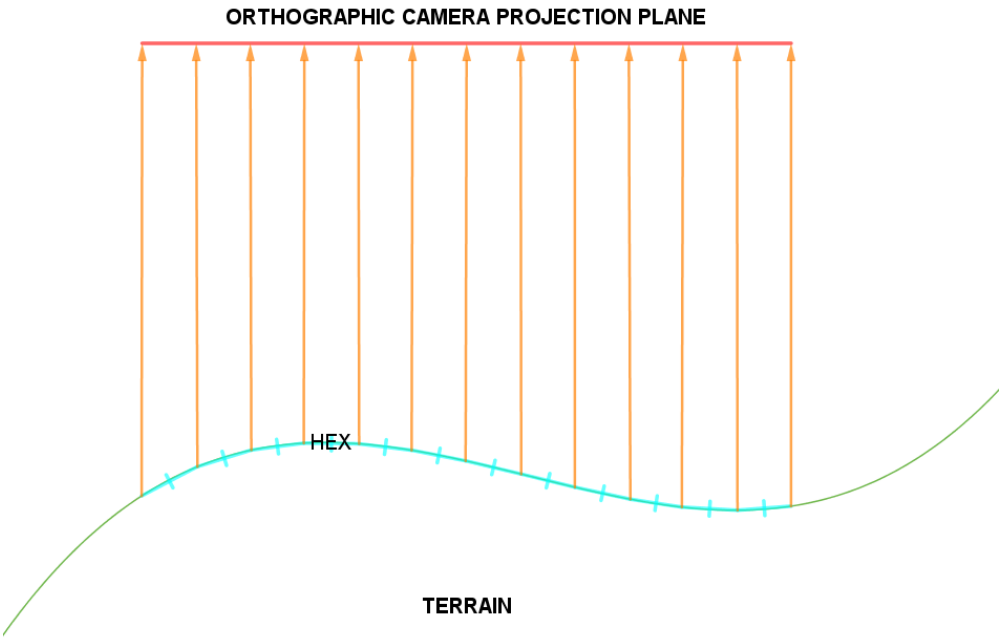

Figure 42. Projection in the orthographic camera

The orthographic camera projection plane can be extended by using the slider provided in the upper left corner of the view. This exposes a wider area to the camera and thus allows to get a greater overview of the land below. The camera can be moved using the WASD keys.

4.3.8.2 Perspective

The book Fundamentals of Computer Graphics [11, p. 73] defines perspective projection as the result of projecting lines from points in three-dimensional space through a single point called the viewpoint. Unlike parallel projection, this type of projection makes objects further away from the viewpoint appear smaller.

Perspective projection is essentially how the human eye translates the light that reaches the lens in the eye. The perspective camera renders images using perspective projection and is the main camera used during gameplay.

5 Testing

Fall was playtested by six people. The objective of playtesting was to determine on a small scale the appeal of the implemented game design and see how people would interact with it. The author believes that *Fall* has combined gameplay elements that have not been age tested by popular games, thus making the development of *Fall* a more unstandardized process. To acquire varied feedback, the game was tested by people with different backgrounds in gaming. Some prefer to play fast paced games, which generally belong to the First Person Shooter (FPS) genre. Others prefer to play strategy games or other games with an overall slower pacing. All testers were required to have at least some prior experience with playing video games. The testers' gaming habits and preferences were evaluated based on their answers to the questionnaires they answered.

5.1 Process

Fall was tested individually with every tester in private sessions. The testers would answer questions in a questionnaire and playtest the game. Learning materials on how to play the game were not included inside the application as part of a tutorial. This was considered, but because the game is not designed to progress by taking the player through levels, the implementation would have required setting up a potentially complex tutorial environment that would have left less time to develop of other aspects of the game. Instead the testers were supplied with an instructions sheet that they were asked to familiarize themselves with before starting the game. Because the players need to be aware of a considerable number of non-standard game mechanics, they were allowed to refer back to the instructions sheet at any time they felt they were unsure about something. This is the second reason why the author opted out of an isolated tutorial session - such a tutorial would have been implemented in a separate environment and it would have been more difficult for the player to go back and find information there during gameplay. It might have even discouraged them to try and find that information. As the testing revealed, whenever the testers needed to find information, they were quickly able to look up the relevant section in the document. Usually they needed small bits of information such as the meaning of the color on a hex. The paper on Brain-Computer interfaces by René van den Berg suggests that people are not opposed to the idea of learning need-to-know information outside of the game environment beforehand, they are just willing to spend less time on it [12]. Although

there are several things to know about the gameplay, the manual itself is structured to provide as much information as possible visually, thus allowing the player to acquire the information faster and not spend too much time on reading.

5.1.1 Testing Environment

Testers played the game on the most recent build. Bugs and performance optimizations were being worked on actively prior to the first testing session. Unfortunately some of the most recent modifications broke parts of previously working code and the newly introduced bugs were first revealed when they started causing issues in the first testing session. The issues experienced during testing are described in chapter 5.2.2.

Four computers were involved in the development and testing (Figure 43). The first test was conducted on **PC1** and the other five tests on **PC2**. Without going into the specifics of hardware setups, the primary hardware in the testing computers is displayed below. **PC3** and **PC4** are the computers used while developing. Their performance is not reflected in the testing results described in this section, but may be referenced elsewhere in this work.

Figure 43. The computers used in development or testing

	PC1	**PC2**	**PC3**	**PC4**
GPU	GeForce GTX 980 Ti / 6 GB GDDR5	GeForce RTX 2070 8 GB GDDR6	GeForce GTX 980 6 GB GDDR5	GeForce GTX 1060 6 GB GDDR5
CPU	Intel I7 6700	Intel I9 9900K @3.6 - 5.2 GHz	Intel I7 6700K	Intel I7 7700 HQ @2.8GHz
RAM	16 GB DDR4	32 GB DDR4	16 GB DDR4	16 GB DDR4

As mentioned earlier, testing was carried out in an isolated area individually with each tester. Testers had no prior knowledge of what the game is like or what the objective of the game is. The hardware tools MSI Afterburner 4.6.1 and RivaTuner Statistics Server 7.2.2 were used to monitor and log performance during testing. MSI Afterburner was used to record the gameplay in a low quality format. During gameplay RAM consumption, frametime and frame rate was logged separately for each tester.

5.1.2 Questionnaire

The questionnaire consists of six questions. Before proceeding to familiarize themselves with the instructions, the testers would answer the first four questions. These first questions were intended to determine the testers' past experiences with gaming and preferences

in game genres. This information would be taken into consideration when assessing their feedback. Next they were provided with a list of features that are in some form implemented in *Fall*, but the testers were asked if they knew of an existing video game that combines as many of these listed features as possible. The last two questions were for giving feedback to the game and were answered after the playtesting session.
The questionnaire is found in Appendix II.

5.1.3 Testing Scenario

The scenario for testing the user experience was the same each time. The objective and process of the game is described in the user manual, but in summary the objective is to hunt all the wolves in the area. The world was populated with ten wolves and their spawn positions were hand-picked before the first testing session. These positions were not changed in-between tests and the every tester played the same scenario, regardless of their skill level.
Some wolves were positioned closer to each-other (in packs). This would lessen the risk that the tester could exploit the *sneak command*, which makes the player character less detectable to the enemy, to easily win the game simply by never exiting this mode. The bow firing range and enemy detection ranges were adjusted accordingly so that the player could not simply run up to the wolves and not alert them. Instead, more often than not, they would have to use the sneak command at some point if they wanted to avoid detection.
The starting position of the player was at a campsite behind an overhanging cliff and the nearest wolves were relatively far from this position (Figure 44). The players would have to locate the wolves by listening to the direction of their howling. This required them to first travel through the woods for a while. This allowed them to first get comfortable with the movement and camera systems before engaging in combat. Despite the long distance between the starting position and the enemies, the volume of the howling was made louder to warn the player to start becoming more cautious with their movements early on.

Figure 44. Enemy locations in the testing scenario

5.2 Analysis

This chapter starts by analyzing how well the game was received by the testers and then provides some statistics on how well the game performed on different systems.

5.2.1 Questionnaire Results

The following subchapters review the information gathered from the questionnaires that were handed out to every tester.

5.2.1.1 Testers

The testers had vastly different backgrounds in their past experience with video games. Majority play games casually and not necessarily every week, whereas one tester barely plays video games on PC at all and instead plays mobile games from time to time. Another tester has spent thousands of hours playing various video games across different genres and has even participated in tournaments for some competitive games such as *Apex Legends*, *Counter Strike: Global Offensive* (*CSGO*), *League of Legends*. See Figure 45 for the distribution of the results.

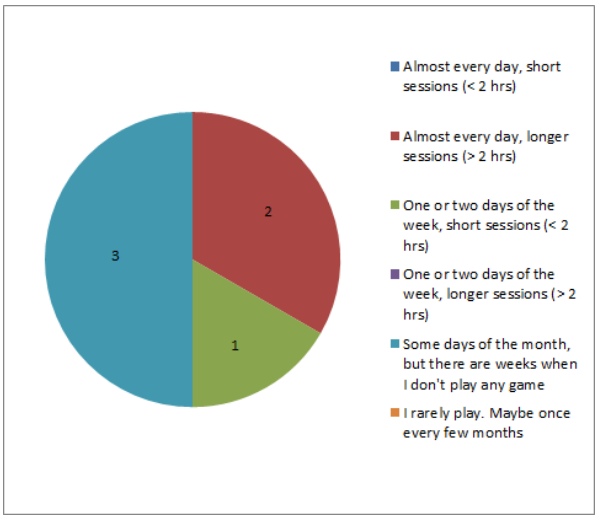

Figure 45. The testers were asked how often they play games

The testers were asked to list some games they have played recently. While the results do not contain games specifically from the tactical role-playing genre, there were mentions of

some games described in chapter 3: *Civilization V*, *Total War*, *Europa Universalis IV*. All but one of the answers share some commonalities with *Fall*, therefore the gameplay style was not entirely new to most of the testers. Their preferences were further explored in question 3, which inquired about the genres they have more experience with. The results are displayed below (Figure 46) and it reveals that the testers seem to prefer fast paced games.

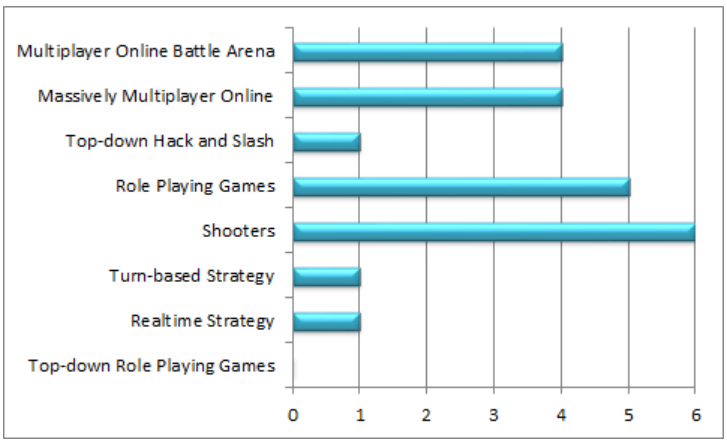

Figure 46. Genres that the testers have most experience in

Games most played by the testers include *CSGO* as one of the fast-paced games. This game was mentioned by three testers. Slower games like *Runescape*, *Hearthstone*, *Minecraft* were also named. *Fall*, like most turn-based games, is an overall slow-paced game and slow games were found among the most played games named by three testers. Regardless of their preferences, none of the testers expressed any major frustration with the slow pacing, although some comments imply that they might have preferred an alternate form of gameplay. Their suggestions are discussed in a later chapter. Two of the testers have prior experience with hex-based games. *Heroes of Might and Magic 3* and *Civilization V* were named as examples.

5.2.1.2 Grading Different Systems

After playtesting the game, each tester was asked to rate their experience with various aspects of the game on a scale from “1” to “5”. Everywhere but the *difficulty* rating the “1” is “terrible” and “5” is “great”. The difficulty is instead evaluated with “1” being “impossible” and “5” is “too easy”. The results are reflected on Figure 47.

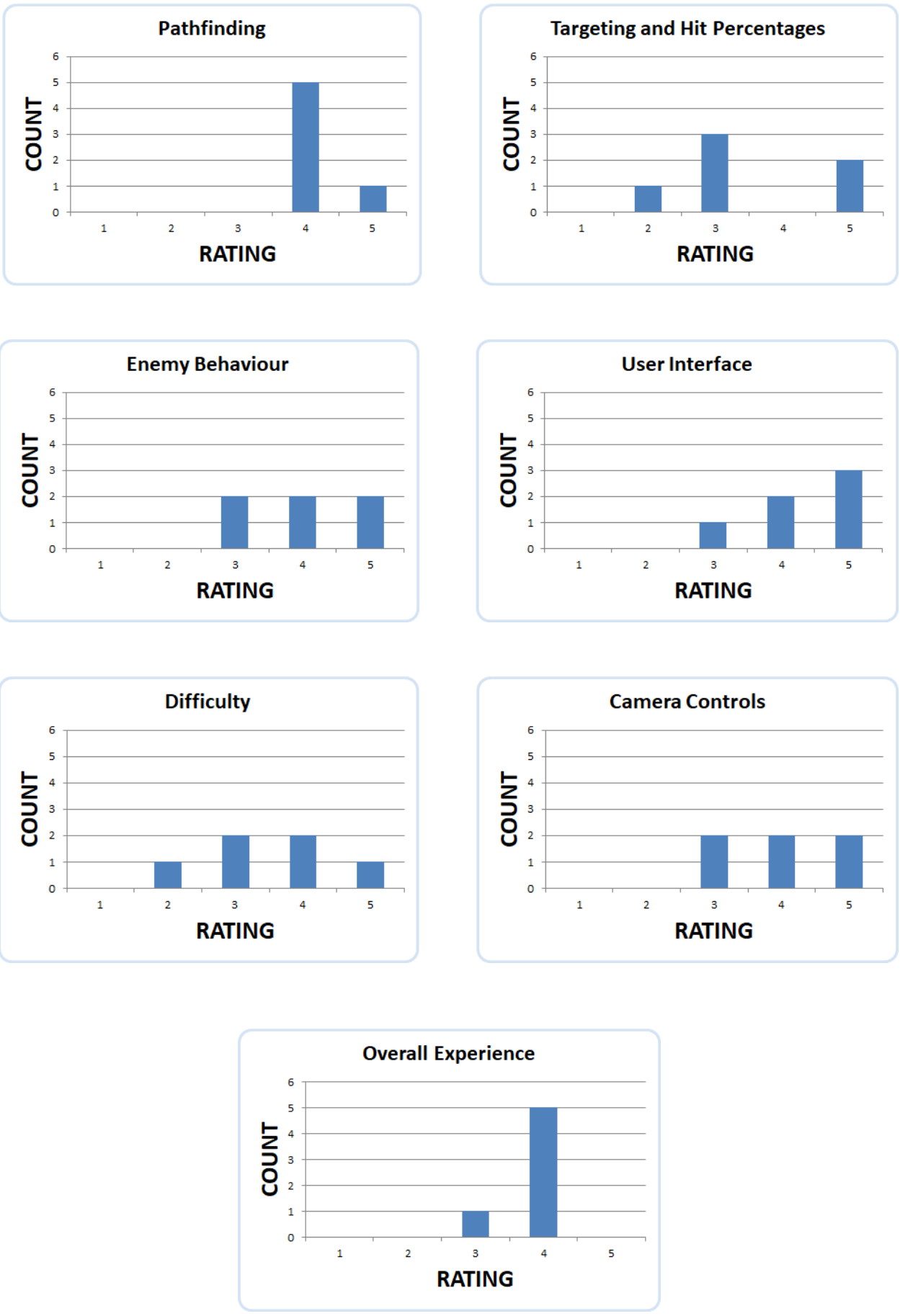

Figure 47. Rating results

The testers were asked to think of items to criticize and insert their criticism in a comment next to the rating for each system. This chapter will now use the ratings and comments provided to analyze how satisfied the testers were with these systems.

The testers were overall satisfied with the **pathfinding**. Three testers would have preferred a faster movement animation, particularly when sneaking. Sneaking has to be slower than regular movement however, otherwise there would be no incentive not to sneak. One comment mentioned that sometimes selecting the hexes did not feel accurate. This could happen when the player does not move the camera around enough and instead leaves it viewing some specific area while sending the player character off into the distance. This behaviour was unanticipated and thus the camera was not configured to trail after the player if the player character wanders far from the camera. On the other hand the more experienced testers would adopt the **camera controls** faster and were looking around the environment more often.

Otherwise the camera controls received positive feedback. Three testers described it as "smooth". Some would prefer to be able to move it even further from the player character. One commented that it was bulky at start, but got used to it quickly. Another tester said that they experienced some difficulty regulating the height and that the treetops often obscured vision too much.

The testers seemed to grasp the logic behind **hit percentages** well and this did not cause problems for anybody. What did cause confusion however is the fact that they have to use the sneak command to be able to kill enemies effectively. Many testers made the mistake of thinking that the wolves will always die in one shot when in fact the bonus damage is applied only when the enemy has not been alerted. This was mentioned in the instructions, but the author concludes that the **targeting** mechanic that was not clear enough and a feature that might help would be a pop-up message upon impact. The most experienced player in the group however quickly found ways to use the system to their advantage and instead of going after the wolves, opted to stay in sneak mode and wait for the wolves to approach the player and then use the bonus damage modifier to take them out with one shot.

Enemy behaviour (AI) was not developed to be sophisticated for this thesis, but the testers were asked for feedback on whether they felt like the wolves were too fast or too slow or if they did too much damage to the player. Some agreed that the wolves dealt too much damage (the player character can take four hits before they die), others said that the wolves are able to move too far. A greater issue was that two testers had problems distinguishing

the red hexes in attack mode from the red hexes in move mode. They would either attempt to issue move commands while in attack mode or simply not know that stepping on a red hex when issuing a move command would alert a nearby enemy and cause them to attack the player. This led to some early deaths. One tester had problems hearing the wolves, which combined with their tendency to not use the camera enough led them on a long detour around the world, unintentionally avoiding all the wolves in the process. The camera needs to be moved during the audio clip for the player to hear the howling volume change in the right or left headphone speaker. The audio clip was long enough for the other testers to notice the effect.

The **user interface** also had one problem during testing. The instructions sheet informed the player that they need to click on the "move command" icon to start issuing movement commands and also that the hourglass icon will light up when they have entered combat mode. At the start of the game the player character is always in exploration state and can immediately start issuing move commands without clicking on the icons. Regardless, before moving, several testers would click on the move command icon first and by doing so triggered a glitch that confused some of them: as they started issuing the move command, the hourglass falsely lit up as well and made the tester think that they were already in danger even though the nearest enemy was far away. This glitch however turned out to be a boon for testing because it revealed that the icon is not a particularly good indicator of game state changes because only one in three testers expressed concern about this. Instead, the game should display a more obvious indicator such as a large banner that briefly appears in the middle of the screen to let the player know when they enter combat state. Otherwise the user interface was described as easy to understand due to its simplicity. One tester recommended adding tooltips to the icons.

The testing scenario was intentionally somewhat **difficult** because the objective of testing was not specifically to reward the tester for beating the game, but rather observe how the game is being played over a course of time and derive conclusions about different playstyles. Unfortunately the first testers did not have a chance to play long enough due to stability issues that were present in the first sessions. These issues are described in chapter 5.2.2.1 below.

5.2.1.3 Suggestions

In addition to the suggestions mentioned in the previous chapter, the testers were asked if they were to add something new to the game or change anything about existing features,

what it would be. This question was intended to get a general understanding of how interesting the game was for the testers and determine whether the game was even comprehensible.

The testers offered both constructive criticism and ideas for future development. Some of the more interesting suggestions or criticism are as follows (paraphrased):

- More clues or hints to help track the enemies more effectively
- Add traps
- The world is too large for hunting wolves (lots of "empty exploration")
- Add an interactive ingame tutorial
- More enemies and melee combat
- Improved animations
- Improved decision-trees for the wolves
- Lower the maximum hit chance to 80-90% (100% was too easy)
- Atmospheric effects that include better lighting, textures, volumetric fog etc.
- The player character needs a way to regenerate health
- Weather effects

Those that responded complimented various aspects of the game and said that they enjoyed the experience overall.

5.2.2 Stability Observations

This chapter documents some performance and stability-related issues experienced during the first two testing sessions and describes how the issues were resolved for the next tests.

5.2.2.1 First Testing Session

Prior to the first testing session it was known that the amount of *Random Access Memory*[84] (*RAM*) reserved by the application is quite large even upon first entering the game.

At some point during playtesting the game froze and the author had to intervene. At the time the freeze was believed to be caused by the recording, but after a while the game would freeze again. This time the freeze was fatal for the application because a wolf that was being moved by a coroutine was caught in-between different states and got stuck in an animation loop.

The tester agreed to try again. Unfortunately they were not able to play long before the game would freeze again and the playtesting session was concluded prematurely. Upon

[84] https://en.wikipedia.org/wiki/Random-access_memory

analyzing the results later the potential cause for the issue was identified as a *memory leak*[85]. Using Unity Profiler the source was tracked and found to be originating from the way hex materials were being handled. Every hex had two materials and each time the highlight of a hex needed to change, both or one of its materials were swapped for the new material accordingly. The memory leak originated from the way Unity handles instanced material swapping. Every time a material was swapped, the new material was added to the memory as a new instance, but the previous material instance was not removed from memory immediately. Over time the number of material instances grew and occupied more memory space. Because the highlights were similar appearance (unlit colour), the material swapping approach could be discarded in favour of an exposed shader parameter. By changing the colour and alpha parameters of the hex materials without replacing them with new materials every time, this leak was resolved. Having two material instances on every hex is not an optimal solution though and could be optimized further to reduce memory consumption and relieve load on the hardware. After the final testing session the memory handling was optimized such that the hexes no longer have instanced materials and now use type-based shared materials (movable, highlighted, 25% hit chance etc.).

5.2.2.2 Second Testing Session

The second testing session had three testers using PC2. This session did not pass without issues either. Despite the issues, the participants were able to play long enough to become familiar with the game and return both positive comments and constructive criticism.

The **first testing** in the second session was conducted on a new build that included a fix for the memory leak from the previous testing. This session revealed a game-breaking bug that was also present in the first build, but did not occur then because the tester did not attempt to move onto any of the hexes that would introduce this problem.

The bug that occurred in this session was caused by hexes that were not supposed to be navigable for any character in the game (blocked by the environment). The bug was introduced by an optimization made days earlier. At the start of the game the hexes use an on ground capsule collider to identify if the hex is blocked by the environment. The capsule triggers the process that disables such hexes, but the capsules were not correctly positioned on the ground and therefore the hexes were not disabled. The tester happened to click on one of these hexes and the application froze because the path calculations began

[85] https://en.wikipedia.org/wiki/Memory_leak

looking for a path from the whole map, including areas outside of the movement range, but it would never find the path. The process would end when every node in the graph is marked as VISITED. The shortest path calculation did not check against this either, since it would incur some performance loss in pathfinding speeds while this situation can only happen due to a bug such as the one described earlier.

Once the first testing of the second session concluded, the bug was found and fixed on site and a new build was created for the **second testing of the second session**. Everything worked well until the tester ran into the first wolf. Attacking the wolf immediately crashed the application. Because this would happen every time and being able to attack is essential to progression, the tester was handed the earlier build and was instructed to avoid hexes that contain obstacles. Eventually the application would still crash when an enemy selected an occupied hex as their destination for movement. The bug causing the crash to desktop was then located and fixed on site.

5.3 Optimizations

Initially the world of *Fall* was not intended to be as large as it eventually became. At some point during development the smaller world was abandoned and a new and larger one was created using the methods described in chapter 4.1.1.1. While the increased scope introduced new avenues of research, it also impacted performance, which was less than ideal even on high end systems such as PC3 (hardware listed in chapter 5.1.1).

The final build and project attached to this work have resolved some of the causes behind poor performance seen in testing. Using PC3 the game is now considerably more performant than it was on PC2 during testing. The increase in performance might be attributed to the changes in material handling mentioned at the end of chapter 5.2.2.1. This change reduced the batches count from numbers above 10 000 down to 300-500 on average. This improvement means less draw calls to the GPU, which in itself is fast enough to not cause noticeable slowdowns because of this, but the render state changes triggered by the use of different meshes and numerous material instances (every hex had two instances) impacted performance noticeably[86].

While being more performant now, there is still a lot of performance to be gained from either exploring solutions that optimize the use of many different meshes in a scene at once (for example customizing the render pipeline) or adopting the shader-based approach instead. While according to the profiler in Unity the game does not occupy an absurd

[86] https://support.unity3d.com/hc/en-us/articles/207061413-Why-are-my-batches-draw-calls-so-high-What-does-that-mean-

amount of memory, the fact that hexes can not be batched for rendering is *bottlenecking*[87] the CPU. The file *profiler.png* attached to this project displays the CPU operations when idling ingame (PC3).

Trees were initially a problem as well. The game needed to have a lot of trees that would decorate the world and make it more interesting to explore and combat in, but high numbers of the pine introduced more triangles and vertices to render. The effects of this were mitigated by creating three additional LOD models for the tree. The trees are static and use a shared mesh and materials, therefore they unlike the hexes they are batched. The hexes also have an LOD setting enabled.

Pathfinding was originally based on *Dijkstra's algorithm*[88], but as capsule casting and movement over greater distances in exploration mode were introduced, the algorithm needed to be converted to A* instead. Unlike before, the latency between selecting a hex and seeing the resulting path highlighted is now barely noticeable (observed on PC4 as the weakest hardware setup in the group).

5.4 Testing Summary

The objective of testing was to let a small group of people play *Fall*, collect feedback from them and observe their playstyle and monitor the performance of the game.

Performance monitoring confirmed some suspicions that the author had prior to testing: the mesh-based solution for the hex map needs to be optimized further to reduce load on the CPU, GPU and consume less memory. The main bottleneck is the fact that hex meshes can not be batched: this causes a lot of draw calls to the GPU and consequently slows down the CPU as well. While excess memory consumption has been reduced since testing, it can be optimized further by, for example, releasing distant unused hexes from memory.

Most of the testers had little to no prior experience with games in this genre. Despite this they learned the core concepts quickly and were able to play the game intuitively. The design issues that popped up were related to the user interface or ingame notification system, which in some parts was a bit confusing to them. In other cases the tester did not use the camera as much as anticipated and thus had a lesser experience. These shortcomings combined with technical issues in the first tests shadowed some results, but the overall experience ratings were still positive, averaging at 77%, thus indicating that the game was enjoyable.

[87] https://en.wikipedia.org/wiki/Program_optimization#Bottlenecks
[88] https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Conclusion

The objective of this thesis was to develop a hex-based video game that implements the hex map in an untraditional way by combining new concepts with some familiar elements found in other hex-based games. The familiar elements include turn-based gameplay and tactical combat on the hex map. These elements are commonly used together in hex games featuring arena combat, but in *Fall* they were made to work in a seamless roleplaying open world scenario that features a terrain.

The development of such a game involved an unstandardized workflow as the aforementioned combination of game elements was not found in any other game. Therefore there were no established best practices and while some solutions presented in this work are not the most optimal, they offer value as an example implementation for anyone interested in developing a similar game. Some optimization suggestions for future development are included in the work.

Fall was playtested by a small group of people with varied experience, skill level and preferences in video games. They were asked for feedback on various aspects of the game. While their experience was shadowed by some technical difficulties in the earlier testing sessions and some parts of the design that they were not aware of before making a mistake, their overall feedback to the game was positive and thus the concept of this type of game can be considered viable.

References

[1] Millington, T. Alaric; Millington, William. *Dictionary of Mathematics*. Internet Archive. New York, Barnes & Noble. 1971. Page 235. https://archive.org/details/dictionaryofmat000mill/ (05.05.2019)

[2] Gordon, V. Scott; Clevenger, John. *Computer Graphics Programming in OpenGL with C++*. Mercury Learning and Information. David Pallai. 2019. Page 271.

[3] Tulleken, Herman. *20 Fun Grid Facts (Hex Grids)* Gamasutra. 2014. http://gamasutra.com/blogs/HermanTulleken/20140912/225495/20_Fun_Grid_Facts_Hex_Grids.php (05.05.2019)

[4] Patel, Amit. *Hexagonal Grids*. 2019. Red Blob Games. https://www.redblobgames.com/grids/hexagons/#coordinates-cube (05.05.2019)

[5] Bose, Juwal. *Introduction to Axial Coordinates for Hexagonal Tile-Based Games*. 2017. envatotuts+. https://gamedevelopment.tutsplus.com/tutorials/introduction-to-axial-coordinates-for-hexagonal-tile-based-games--cms-28820 (05.05.2019)

[6] Emrich, Alan. *MicroProse' Strategic Space Opera Is Rated XXXX!* Computer Gaming World. Issue 110. Pages 92-93. Internet Archive. https://archive.org/details/Computer_Gaming_World_Issue_110 (05.05.2019)

[7] Froelings, Lisa. *5 Best Video Game Development Tools for Indie Game Devs*. 2017. DZone. https://dzone.com/articles/5-best-video-game-development-tools-for-indie-game (05.05.2019)

[8] Roy, Sourav. *The Reasons for Unity 3D's Vast Popularity*. 2017. Capital Numbers. https://www.capitalnumbers.com/blog/why-unity3d-popular/ (05.05.2019)

[9] Famularo, Jessica. *What indie developers think of Unity in 2018*. 2018. PCGAMER. https://www.pcgamer.com/what-indie-developers-think-of-unity-in-2018/ (05.05.2019)

[10] Reid, Linden. *X-Ray Shader Tutorial in Unity*. 2018. https://lindenreid.wordpress.com/2018/03/17/x-ray-shader-tutorial-in-unity/ (05.05.2019)

[11] Shirley, Peter; Marchner, Steve. *Fundamentals of Computer Graphics, Third Edition*. CRC Press. Taylor & Francis Group. 2009. Pages 71, 73.

[12] van den Berg, René; *Intuitive Self-Paced Brain-Computer Interaction for Games*. 2009. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.226.6207&rep=rep1&type=pdf (10.05.2019)

Appendices

I Image Examples

Figure 48. **Example use of a non-homogeneous tessellation**. The cells in this case are the territories of the European and Middle-Eastern countries displayed on the game map in ***Europa Universalis IV***

Figure 49. **Example use of a homogeneous tessellation**. Homogeneous (semi-regular) tessellations tessellate more than one type of regular polygon. In the multiplayer survival game ***Rust*** tessellated squares and triangles are an integral part of base building.

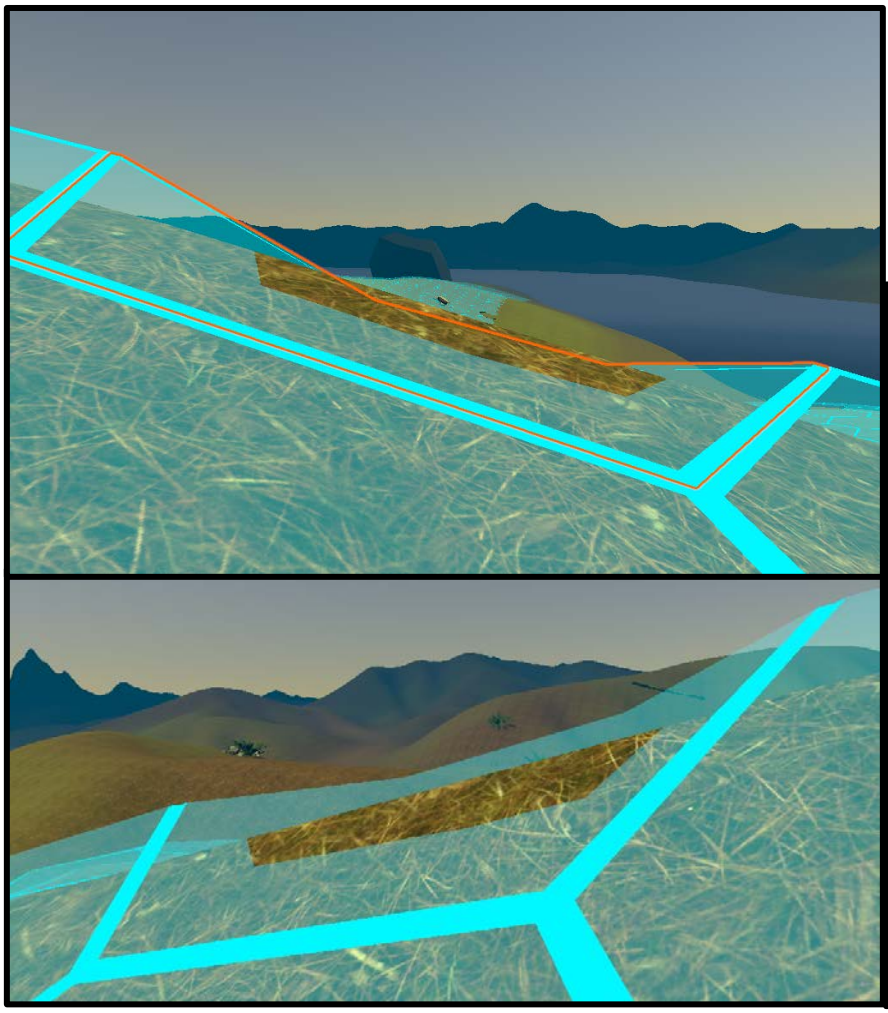

Figure 53. The CPU converts n-gons into triangles and that can cause unpredictable rendering

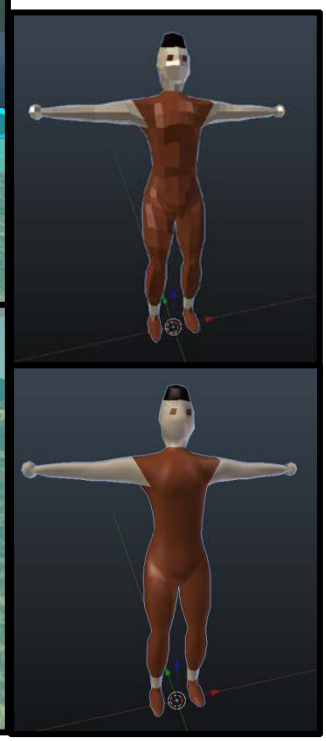

Figure 50. Flat shading (top). Smooth shading (bottom)

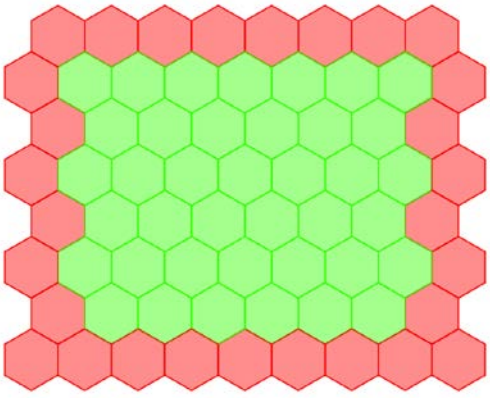

Figure 52. Fixed parameter map. When the player moves around the green area and they reach the border (red), they will know for a fact that there are no more hexes beyond it

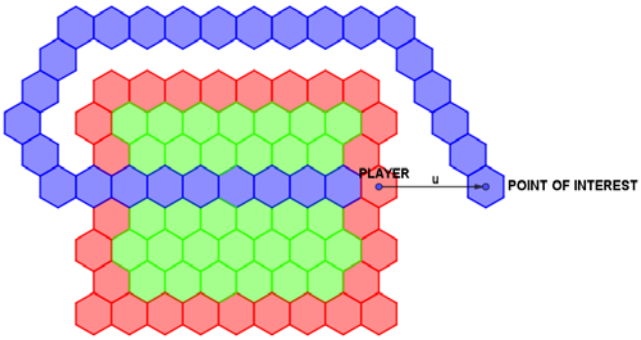

Figure 52. When the player reaches the edge of the map in one area (the current player position depicted), the map does not end there and even though they can not progress to the point of interest directly along the vector *u*, there could be an alternative path, in this example highlighted blue

II Accompanying Files

User manual (**EN**): *Fall – User manual.pdf*

User manual (**EE**): *Fall – Mängu kasutusjuhend.pdf*

Questionnaire: *Fall – Feedback Form.pdf*

The final **build** (the game) is found in the folder "BUILD". No installation is required. Extract the package and run the executable *Hex.exe* inside. The application may take some time to load before it displays anything. Playable performance (above 30 frames per second) on systems inferior to PC4 in chapter 5.1.1 is not guaranteed.

The **project** in its entirety is in a separate folder. Use only Unity version 2018.3.9 to open the project. The project may not work correctly if opened in another version.

The folder **Other** includes a screenshot of the CPU load referenced in the work. Assets used in the work and not made by the author are listed in the file *Foreign Assets Used.txt*. Also included is a demo video.

III Distinctive Features

The distinctive features of *Fall* that were made to work together are as follows:

- Open world exploration
- Roleplaying elements
- Turn-based (combat and movement)
- Tactical
- Hex map on a three-dimensional surface (non-planar)
- Seamless transitions from exploration to combat situations

Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, Oliver Vinkel,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Fall – A Turn-based Role-playing Game on a Hex Map

   supervised by Raimond-Hendrik Tunnel

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Oliver Vinkel
10/05/2019