Bachelor's project

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Sindre Haugland Paulshus

Implementing Optimized Line of Sight based Fog of War in Unity

For the RTS game DwarfHeim by Pineleaf Studio

Bachelor's project in Software Engineering
Supervisor: Helge Hafting
May 2020

NTNU
Kunnskap for en bedre verden

Sindre Haugland Paulshus

Implementing Optimized Line of Sight based Fog of War in Unity

For the RTS game DwarfHeim by Pineleaf Studio

Bachelor's project in Software Engineering
Supervisor: Helge Hafting
May 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU

Kunnskap for en bedre verden

Preface

The project was birthed in the need for a new Fog of War solution for Pineleaf Studio's upcoming Real Time Strategy game DwarfHeim, but became a bachelor project through reaching out to them on my own. The goal was to be able to challenge myself working for a game development studio and be able to produce something of quality in the process. I would like to thank Pineleaf Studio as a whole, but also their CEO Hans-Andreas S. Kleven specifically for greenlighting the project. I also want to thank their CTO Sigurd Mikal O. Murad and their Lead Programmer Kenneth J. Kristensen. Their help has been proved invaluable to the final solution and the main report, constantly suggesting new technologies and providing valuable feedback. I also want to thank my supervisor Helge Hafting, for his feedback and discussions on various topics throughout the project. Lastly, I want to thank my partner, my family, friends and fellow students for their continued support.

Task

The task to be solved for this project is the development of one or more solutions for technical- and design challenges around the game mechanic Fog of War, within the genre of Real Time Strategy games for the game DwarfHeim by Pineleaf Studios [1]. Specific requirements can be seen in Appendix D, Vision Document, and Appendix B, Requirements Document. Additional challenges were performance and optimization, visual experience and clarity in gameplay. It was the performance and optimization part of the task that was to be its secondary focus. The development was to be done mainly in Unity with the programming language C#. See Appendix A for the original task description.

Abstract

This project explores how to implement the game mechanic Fog of War with Line of Sight in a Real Time Strategy game, as well as optimizing it as much as possible. While Line of Sight based Fog of War is nothing new to the genre, implementation and methodology for the mechanic is mostly under wraps. This project hopes to rectify this and create stable ground for innovativeness and research on the topic.

The project came about as Pineleaf Studio wanted to upgrade their current Fog of War solution with a more sophisticated one with Line of Sight for their Real Time Strategy game DwarfHeim. This can be implemented and optimized in various ways and this report's theory gives a foundation for this. It goes through line drawing algorithms, optimization methods and more general topics such as Fog of War and Line of Sight.

Throughout the projects duration, various technologies and methodologies were explored and settled on. Game engine was to be Unity, programming language C#. The development process was iterative and innovative, but no specific framework was selected. The Line of Sight was implemented using the line drawing algorithm Digital Differential Analyzer from the start point to the edge cells of a unit's vision. Optimization methods used were spatial indexing, parallelization and Burst Compiler. Parallelization was implemented using C# Parallel.For and Unity Jobs. Using Unity Jobs opened the opportunity to use their Burst Compiler, which automatically transforms the code to highly optimized native code.

The final result was a performant Line of Sight based Fog of War. The employer was satisfied and all needed requirements were fulfilled, while none of the optional additions were implemented. The development of this result followed its plan broadly, with the exception of a drastic shortening of visual development. This was due to the opportunity to reuse parts of the visual implementation of DwarfHeim's previous Fog of War implementation. Time management during the development was shifted towards the logical development, documentation and research.

The final result is well functioning. One could make the argument for using Bresenham over Digital Differential Analyzer. How much effect that would actually have though, that's up to debate as the solution is also utilizing Unity Jobs and the Burst Compiler. Unity Jobs is very restrictive though, which causes a lot of overhead. Finding a solution for this overhead problem could improve the performance significantly. Even though none of the optional requirements were fulfilled, a way for possibly creating a grayscale of explored areas which could be integrated into the solution was discovered.

In the end, the final result can be viewed as an answer to the research problem. By using DDA between start and edge cells to create each unit's vision, stitching those visions together and applying it as a texture to a plane overlapping the game area, which in turn masks another plan which visualizes the Fog of War. This is optimized by utilizing spatial indexing, C# Parallel.For and Unity Jobs with the Burst Compiler. Future work could include implementing the solution with Bresenham, pre-baked LOS and developing LOS based FOW for other game engines.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

Pineleaf Studio is currently developing the real time strategy (RTS) game DwarfHeim. An important feature for both gameplay and aesthetics in the game is the Fog of War (FOW), the area of the map the player cannot see. The developers had implemented a simple circle-based FOW, but as release was nearing they wanted to upgrade it to a more sophisticated solution with Line of Sight (LOS). This would enhance both the visual experience and expand the gameplay options for players of the game.

While many previous RTS games have employed this technique, the methodology behind its implementation is very much a trade secret. Anecdotes from developers and some open source projects exist, but step-by-step guides or articles on the subject are not available. The methods to make it does exist however, so that will be the task of this thesis. Hopefully, when this report is done, it may act as an entry point into the field of Fog of War in game development.

This thesis is written by a single student for his bachelor's degree in Software Engineering at NTNU, the Norwegian University of Science and Technology, during the first semester of 2020.

1.2 Research Problem

The main problem to solve in this project is to implement a LOS based FOW and document how this is done. Additionally, there are many areas one could focus at during the development of such a solution. There is fog aesthetics or how it affects gameplay, both valid focus points. However, the most basic requirement for any solution to this task is that it runs well on targeted consumer hardware. As such, the solution itself must have negligible impact on game performance. It is this basic requirement which is the secondary focus of the thesis, as formulated below.

"How can a Line of Sight based Fog of War solution for a Real Time Strategy game that maintains desired game performance be implemented in Unity?"

This research problem consists of two parts. Firstly implementing a line of sight based fog of war, and then also optimizing it so it maintains desired game performance. These two parts are undoubtedly connected however, as specific implementations may lead to better performance.

1.3 Report Structure

This report is structured after NTNU TDAT3001's Bachelor's Thesis main report template, last updated 05.12.2018.

As seen in the project contents, the report is broken into six main parts. These can be seen in Table 1.

Introduction	Introduces the project, its background, problem and report structure.
Theory	Details the theory behind the project.
Choice of Technology and Methods	Explains what choices were made and why in relation to technology and methods used in the project.
Results	The scientific, engineering and administrative results of project.
Discussion	A discussion based on the results in relation to the research problem, requirements and choices made during the project's course.
Conclusion and Future Work	Conclusions drawn on the research problem and the project's requirements based on the discussion and results. Includes recommendations for future work.

Table 1: The Report Structure.

1.4 Abbreviations, Acronyms and Definitions

Abbreviation, Acronym or Definition	Description
FOW	Fog of War
DDA	Digital Differential Analyzer
LOS	Line of Sight
RTS	Real Time Strategy
Cell or Tile	The smallest discrete portion of the game world. Usually square- or hexagon-shaped [2, p. 280].
World	The total area in which the game takes place. Internally it is usually represented by a 2D array of cells. Also called a map [2, p. 280].
Player	A human or computer individual who controls a set of units [2, p. 280].
Unit	A game object controlled or owned by a player. Can be movable or immovable [2, p. 280].

Table 2: Abbreviations, Acronyms and Definitions.

2 Theory

2.1 Real Time Strategy Game

Real-time strategy games (RTS) is one of two subgenres of strategy games, the other being turn-based strategy games. Moss defines RTS as a game genre that involves base building and/or management, resource gathering, unit production and semi-autonomous combat. All this conducted in real time, with the game's goal being gaining or maintaining control of strategic points of the map, such as resources or command centers [3].

Some view RTS games as simplified military simulations, as by Buro in his 2003 poster. The struggle for resources between players, building and maintaining an economy, raising and controlling armies and waging war against opponents in real time is reflected in real militaries and war [4].

Incidentally, Buro calls for AI research into RTS. Today, RTS games have become a fertile ground for AI and machine learning research [5]. Popular games like StarCraft 2 have garnered much attention in the field recently [6][7].

Figure 1: Media from the popular RTS "StarCraft 2" by Blizzard Entertainment [8].

2.2 Fog of War

Fog of War (FOW) is a game mechanic usually featured in RTS games where specific and dynamic game space information, like enemy units' positions and actions, is hidden when the player has no units of their own in range. In other words, the player can only see parts of the game space where they have friendly units. This is called the player's vision [9].

> "War is the realm of uncertainty; three quarters of the factors on which action in war is based are wrapped in a fog of greater or lesser uncertainty."
> -On War, Carl von Clausewitz [10, p. 101].

The term "Fog of War" has its roots in the military book On War by Carl von Clausewitz, written in 1832. In it he describes among other things the unreliability of information, and the importance of intelligence in war. While the true meaning behind his words and arguments are debated till this day, it works well in this context [11]. It is important to note that term "fog" is used as a metaphor for the hidden and uncertain in war.

This all parallels to RTS games in general being viewed by some as military simulations. In that case, the FOW is a part of the simulation, an attribute of real war transferred to the game. It is transferred in a literal sense, as the fog that envelops what the player cannot see, as well as in a broader sense of information deprivation.

A related game mechanic to FOW is *mirages*. Mirages are static ghostly representations of enemy units appearing inside the FOW. They represent enemy units as they were the last time they were seen. In some implementations, these mirages remove themselves when their corresponding unit reappear for the player [2, p. 286].

2.3 Line of Sight

Line of Sight (LOS) is the physical phenomena of having a direct and open line from a viewport, like an eye or camera, to a given object. LOS between the viewport and any given object may be blocked by the environment. In a broader sense, LOS is all that is viewable from the viewport [12, Chapter 2].

Many genres of games adopt this logic as a game mechanic, also RTS games. In RTS games it is generally implemented as a viewable area around the player's units with a given radius, which then may be blocked by terrain and other structures in the game. This is implemented in coordination with the FOW [2, p. 280].

2.4 Game Engine

A game engine is not clearly defined concept, argues Gregory, but rather a blurry line between the game itself and its core software components. He thinks all imaginable game engines falls somewhere on the range between a game engine that can make any game and a game engine that can only make one game. This is visualised in Figure 3 [13, Chapter 1.3].

Gregory argues that data-driven architecture is the real difference between a game and a game engine. A game has specific and hard-coded data, that is to say models, images, logic or game rules. On the other hand, a game engine is meant to be used as a foundation for many different games without extensive modification and as such contains tools and components that can be used and configured to each game's needs [13, Chapter 1.3].

Figure 2: Fog of War with Line of Sight as seen in ”StarCraft 2” [8].

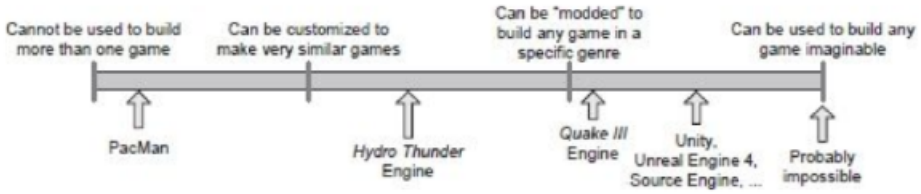


Figure 3: A range of all possible game engines [13, Chapter 1.3].

Inherent to this range of game engine possibilities there is a trade-off. A game engine tailored to make a given genre of games, will invariably be less suited to making other genres of games in. It is certainly not impossible, but it is not optimal. With different genres of games having many different needs in terms of components and mechanics, there is a clear trade-off for general purpose game engines. The more general purpose a game engine is, the less optimal it is for making and running a particular game on a particular platform [13, Chapter 1.3].

Most modern game engines are somewhat genre specific. A fighting game will have different needs than a massively multiplayer online role-playing game (MMORPG), and as such their game engines will be very different. Even so, there are common denominators between most game engines. All games, regardless of genre, require for example some form of user input, graphics rendering, heads-up display (HUD) and GUI with text, and an audio system [13, Chapter 1.3].

In addition, some modern game engines feature a physics system. A physics system allows the game to simulate physics, or more accurately rigid body dynamics, during gameplay. Collisions detection, forces and torques are some usual features included in a physics system [13, Chapter 1.3].

2.5 Digital Differential Analyzer

The digital differential analyzer algorithm, DDA for short, is the simplest line drawing algorithm available. It has its roots back to analogue computers called Differential Analyzers, which aimed at solving the differential equation $y' = m$ [14, p. 196].

DDA calculates pixel positions along a given line by using the equation $y = mx + b$ for a straight line. m is the slope and b is the y-axis intercept [15]. In computer graphics, this line may be given as a vector with start and end points. That is where the DDA algorithm comes in to convert the vector to a raster (called rasterization) [16, p. 232]. A raster is a grid of dots that form an image, which can be stored in a bitmap or displayed as pixels [17]. Using the algorithm, we can draw a raster line between the start and end points.

Figure 4 shows pseudocode of the DDA algorithm, based on an example by McKinney [15]. Example has been modified to resemble more modern programming. In the original sample "trunc(x)" was used to truncate the values to integers, but that has been replaced with truncating them by casting to int. Both achieves the same purpose, which is removing the fractional part of the value.

```
DDA (x1, y1, x2, y2)
    int length
    float x, y, xincrement, yincrement

    length = abs(x2-x1)
    if (abs(y2-y1) > length)
        length = abs(y2-y1)

    xincrement = (x2 - x1) / length
    yincrement = (y2 - y1) / length
    x = x1 + 0.5
    y = y1 + 0.5

    for (int i = 0; i < length; i++)
        PutPixel ((int) x, (int) y)
        x = x + xincrement
        y = y + yincrement

    PutPixel(x2, y2)
```

Figure 4: Digital Differential Analyzer Line Algorithm pseudocode.

It is clear there are two cases the program in Figure 4 can take. If the value of the slope is less than 1, it will increment the x value by 1 and the y value by the slope. If the value of the slope is more than 1, it will increment the y value by 1 and the x value by the reciprocal of the slope. The pixel is then selected by adding 0.5 to the values and casting them to int, which in essence is the same as rounding to the closest integers [15].

Notice also that the algorithm uses floating-point calculations, absolutes and divisions which slows down the algorithm. This makes it unsuitable for certain uses, like for computer graphics devices, where there are thousands of lines that needs to be drawn every frame. However slow it is, it is

simple and accurate, selecting the pixel that is closest to the actual line [15].

2.6 Bresenham's Line Algorithm

Bresenham's Line Algorithm is one of the earliest algorithms used in computer graphics. First written about by J. E. Bresenham in 1965, this algorithm circumvents the slowness of the DDA by requiring no division and no floating point calculation [18]. Only employing integer values and multiplication with 2, which can be done by left shifting the value [19].

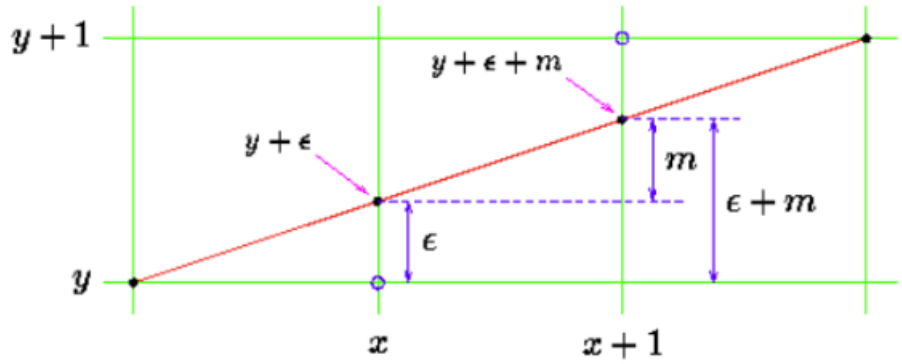


Figure 5: Bresenham's algorithm plotted explaining the values. y is the line, e is the error and m is the slope [19].

It works by using an value e to control plotting, where e is the error between the real line y value and the selected y value, as seen in Figure 5. This means the real y value is $y = y + e$, and e will range from -0.5 to just under $+0.5$. When incrementing x to $x+1$, the real y will increase by the slope m to $y = y + e + m$. We decide whether to plot $(x+1, y)$ or $(x+1, y+1)$ by looking at difference between the new real y and y. If $y + e + m < y + 0.5$, we plot $(x+1, y)$. Otherwise we plot the other. Then we update the error accordingly, which will rise with m to $e = e + m$. If we plot $(x+1, y+1)$, the error will actually be $e = e + m - 1$ to account for the increment in y. These values can be seen in Figure 5 [19].

$$1)\ y + e + m < y + 0.5$$

$$2)\ e = e + m$$

$$3)\ e = e + m - 1$$

Notice that this method still uses floating point values in the plotting test. To solve this, Bresenham removes y and multiplies both sides of the test by Δx and then by 2. Note that the slope m can be written as $m = \Delta y / \Delta x$ [19].

$$1)\ y + e + m < y + 0.5$$

$$1)\ e + m < 0.5$$

$$1)\ e + \Delta y/\Delta x < 0.5$$

$$1)\ e\Delta x + \Delta y < 0.5\Delta x$$

$$1)\ 2e\Delta x + 2\Delta y < \Delta x$$

He then substitutes a new error value e' for $e\Delta x$. The test then becomes $2(e' + \Delta y) < \Delta x$, which is integer-only. The new error e' will update based on x, so to find the new e' update rules he just multiplies them with Δx. Note also here that m can be written as $m = \Delta y/\Delta x$ [19].

$$2)\ e = e + m$$

$$2)\ e\Delta x = e\Delta x + m\Delta x$$

$$3)\ e\Delta x = e\Delta x + m\Delta x - \Delta x$$

May be written as:

$$2)\ e\Delta x = e\Delta x + \Delta y$$

$$3)\ e\Delta x = e\Delta x + \Delta y - \Delta x$$

In e' form:

$$2)\ e' = e' + \Delta y$$

$$3)\ e' = e' + \Delta y - \Delta x$$

With that Bresenham's algorithm is complete. See pseudocode in Figure 6 based on example by Flanagan. This example only applies to slope value of $0 \leq m \leq 1$ [19].

```
Bresenham (x1, y1, x2, y2)
    dx = x2 - x1
    dy = y2 - y1
    e = 0
    y = y1
    for (x = x1; x < x2; x++)
        PutPixel (x, y)
        if (2(e + dy) < dx)
            e = e + dy
        else
            y = y + 1
            e = e + dy - dx
```

Figure 6: Bresenham's Line Algorithm pseudocode.

2.7 Spatial Indexing

Spatial indexing is the technique of accessing spatial data efficiently using a data structure called spatial index. This is in contrast to a sequential scan of the data, where worst case scenario is that

you have to check every data point. Using a spatial index, one may look for relevant information within an area of interest, filtering out all redundant, outside data. Spatial data or spatial objects are multidimensional data such as points in space, lines, polygons or circles, most notably used within the field of Geographic Information Science [20].

An important concept in spatial data is the Minimum Bounding Rectangle (MBR). In a 2D space, MBR is defined by two points $(xmin, ymin)$ and $(xmax, ymax)$ that envelop the data. When performing an operation, we first use the MBR and filter everything outside it out. This way the search area is reduced and operation speed increased. More complex methods for spatial indexing do exist, such as Fixed grid index, R-tree, Quadtree, KD-Tree and Geohash, but they all use MBR in one capacity or another [20].

Fixed grid index is one simple and common method that partitions the space into a 2D grid with equal cell size. This is stored in an array where each index is associated with a list of spatial objects that intersect or overlap a particular cell. The MBR is in this case used to determine which cells an object overlaps or intersects [20].

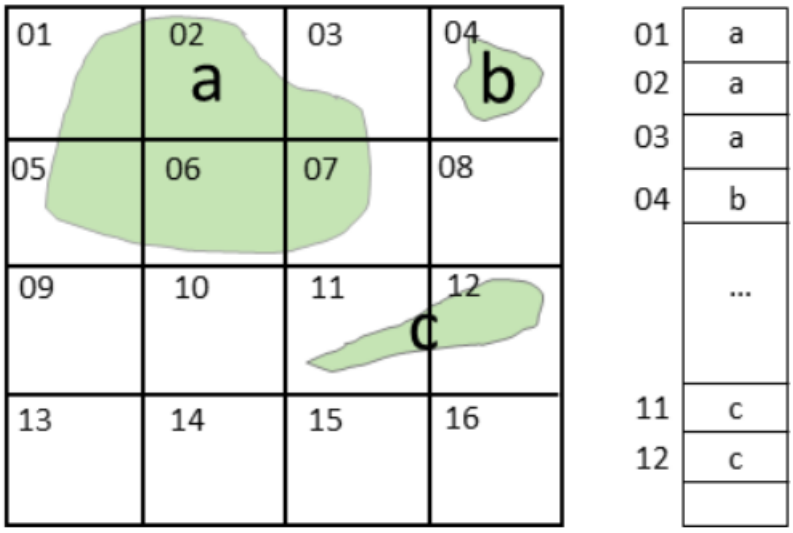


Figure 7: Example of fixed grid index method in use [20].

2.8 Burst Compiler

The Burst Compiler, hereby Burst, is a technology developed by Unity for their game engine. It takes Unity jobs, structs of a set interface created with multithreading in mind, as input and generates highly optimized native code from Intermediate Language (IL) / .NET bytecode. It optimizes the code for the targeted platform automatically. To use Burst, one only needs to add the attribute ”[BurstCompile]” to the job, as seen in Figure 8. This way, developers can efficiently create high-performance C# code [21].

Using Burst comes with some constraints however. It only supports primitive types such as bool, byte, short, int etc. Most notably it does not support strings or regular arrays, though arrays can be used with the native container $NativeArray < T >$. Burst does allows its own specialized vector

types however, such as float2 and float3, which holds 2 and 3 floating-point values respectively [21].

```
[BurstCompile]
public struct MyJob : IJob
{
    public void Execute()
    {
        //Behaviour here
    }
}
```

Figure 8: Example of how the Burst Compiler is enabled.

3 Choice of Technology and Methods

3.1 Game Engine and Programming Language

As the FOW solution is to be integrated into an already existing game, which is currently under development, there was no room for negotiation on game engine and programming language. The solution was to be made in the version Unity used by the employer, version 2019.3.0f3, and scripted using C#. This was neither a surprise nor inconvenient. I had ample experience in both and was ready to tackle new problems using them.

3.2 Development Process

The development process was influenced by the development process of DwarfHeim. Weekly standup meetings with the DwarfHeim team were had where progress and plans were laid out briefly. The development process was also inherently iterative. This is because Unity features a "play" option, which swiftly and easily lets the developer test play the game. This feature, alongside writing to the console, was used extensively to ensure the quality and functionality of the solution as it was developed.

3.3 Calculation of the Vision

When creating a FOW it is noteworthy that the visual fog itself is the inverse of what the player can see, it is the inverse of a player's vision. That vision is gained from the player's units. Therefore, a clever method of creating the FOW is by calculating its inverse, the player's vision.

There is also the possibility of creating the FOW directly, by iterating over all the cells that make up the world grid, and for each cell checking if one or more units' view distance overlap with the cell. To check for units efficiently, one could use spatial indexing. The problem with this solution is that it does not lend itself to LOS very well. You would have to draw a line between the units and the cell until at least one has a direct path and has vision to the cell. If there are several units

in range, but none have vision due to walls or other obstructions, another issue arises. The cell will draw a line for each unit in range every update, even if it is behind some obstacle. However, this solution does have the advantage that if multiple units see the same cell, which is common in RTS games, as you only need one of the units to draw the line from them to the cell for it to light up, which pulls down the average operation time.

Another issue with this solution is how to skip inactive units. A common occurrence during gameplay is that only a few units have moved during a given time frame. The FOW needs to be updated to account for that, but the impact of several units will be unchanged. This could possibly be solved by having the cells know which units have vision to them, and having the units know which cells they have vision to. Then that knowledge can be reset upon unit movement. When iterating over the cells, each cell would first check their unit list. If it is not empty, this cell is lit. If it is empty, search for nearby units as previously explained. In this case, the solution would be better if all nearby units draws a line to the cell, even if another unit already has vision there. However, this forgoes some of the advantages of the solution to begin with.

For this project, the FOW is not created directly. As hinted in the first paragraph, the solution is based around calculating each unit's vision separately. This is done by first calculating the edge cells of a unit's vision and then drawing lines from that unit's starting cell to its edges. The line stops if any obstacles are encountered. Once calculated, the unit knows which cells it has vision to, and it only has to recalculate if it moves. All the units' visions are then totalled and the FOW is set accordingly.

Lastly, I just want to acknowledge that there is another method to create LOS based FOW. This is done by raycasting to the corner endpoints of obstacle polygons. This method was not explored during this project, but it may be the most prevalent one [22].

3.4 Drawing Lines

The way to simulate LOS used in this project is by drawing lines between the unit's start cell and the target edge cell. One of the easiest ways to do this is by employing raycasting. Firstly, cast an amount of rays in all directions with length equal to the unit's view distance, and each ray will return its hits in order. Then iterate through the hits from start to finish and stop if an obstacle is found. While this solution is simple, it is very performance heavy. Raycasting in itself may be an optimized operation, but it does not scale well with the amount of units present. Each unit needs to cast an amount of rays to cover their vision every time they move. In addition, each cell must have a physical presence with a collider to be able to be hit by the ray. Raycasting also has another major flaw. As each ray travel from the unit, there will be an increasing distance between a ray and its neighbour rays. If this distance becomes too large, the rays may miss cells which should be in the unit's vision. This will be a problem with larger view distances, but can be offset by casting even more tightly packed rays. This would unfortunately increase the performance load. Therefore, it was decided that raycasting was not a good solution.

However, there are others way of finding out which cells a unit sees, namely drawing lines to their vision's edges using mathematical algorithms. As depicted in the report's theory, there are, among others, two simple algorithms for drawing raster lines: DDA and Bresenham. Note that there

are more modern line drawing algorithms which are far more popular than these two, like Wu's line algorithm which features antialiasing [23]. Antialiasing is, in short, the smoothing of jagged features caused by discrete sampling [24]. For this project, however, only DDA and Bresenham were considered. Of those, DDA was chosen, despite its use of floating point calculation and division. This was mainly because it was easier to implement, but also because in this use case the difference in performance impact is negligible. In a computer graphics device you would choose Bresenham over DDA because of its inherent speed, but also because integer-only math is easier to implement on an minuscule level. For high level usage like this, however, the impact was considered to be insignificant. This is especially true when using the Burst Compiler. It would drastically increase the performance of the line drawing algorithm to the point where optimizing the algorithm itself would only give minor results.

3.5 Iteration over Units and Cells

An issue encountered when developing the solution is how to effectively iterate over units and cells. When you have hundreds of units, it is best if you can narrow down which units that needs to be updated at any given time. If a unit on one side of the map moves, the FOW on the other side of the map does not need to be updated. Only the FOW within the MBR of the unit's current position and previous position needs to be updated. In the solution, spatial indexing is employed for that very reason. Limiting the amount of units and cells that needs to be iterated over undeniably leads to better performance in general.

3.6 Parallelization

When optimizing code, one common way to increase performance is to utilize parallelization, that is running the code on several threads and taking advantage of multiple CPU cores. In C# this can be done in a few ways. C# supports creating threads directly from classes' static methods using their System.Threading package [25]. However, this method was not explored during this project, but is possibly the most common way to implement parallelization with C#.

Another way to implement parallelization in C# is through the use of Parallel.For and Parallel.ForEach methods, available from the System.Threading.Tasks package. It allows the developer to quickly parallelize a for- or foreach-loop, assuming the loop does not need to break or get canceled, and that each iteration of the loop is independent from other iterations [26] [27]. In the project, both Parallel.For and Parallel.ForEach was experimented with, and in the final solution several Parallel.ForEach are used. It is a simple and effective way to iterate through an array or list.

Finally, Unity has created its own packages for parallelization. One of their key components is the Jobs system. When using Unity Jobs you can use the interface IJobParallelFor, which is a parallelized for-loop, similar to the Parallel.For. The advantage this has, compared to the latter, is that by using a Unity Job, you also get access to the Burst Compiler [28].

4 Results

4.1 Scientific

The solution that has been created is a LOS based FOW for an RTS game, which is able to run with relatively little impact on game performance. The solution's overarching structure can be seen in Figure 9.

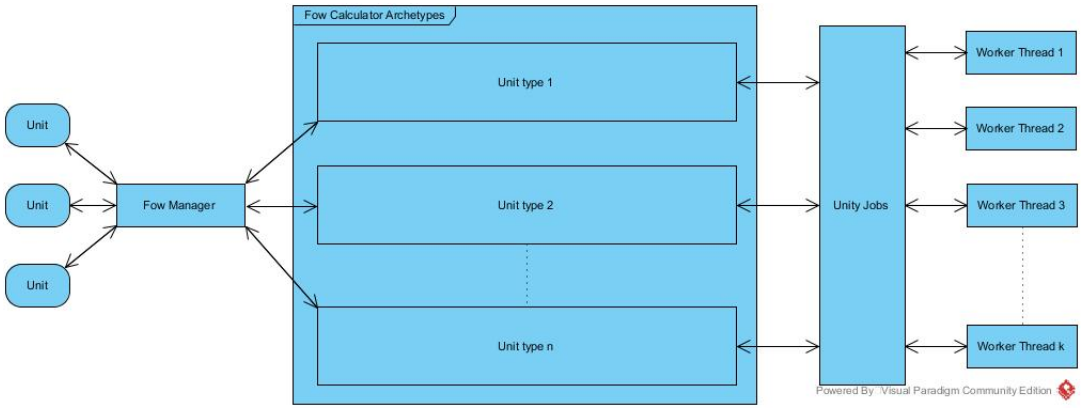


Figure 9: The overarching architecture of the solution, as seen in Appendix C.

The FOW is created through a series of steps. When a unit spawns or moves, it will send an update request to a central FOW manager, which adds the unit to its unit update list. This manager updates the FOW on a given, configurable timer, as long as it has gotten a request to update. This timer limits the amount of FOW updates that are done per second, and it also serves to batch together several units' update requests. The manager then aims to do two things: Set a Color32 array, which represents the inverse of the FOW, with the appropriate colors and then create a texture from that array to apply to a plane which overlaps the game area. This plane is invisible to the player however, but is used to mask another plane, using a specialized shader, which is the actual visible FOW.

Setting the Color32 array is where most of the heavy lifting is done. To do this, the FOW manager needs a list of indexes corresponding to the units' totalled vision. This is, as mentioned before, done by calculating each unit's vision. Each unit type has a corresponding archetype (also called template) of how their vision is calculated, each with their own view distance and a maximum amount of resources or walls their vision can penetrate. The manager groups the units in its unit update list into these archetypes. Then each archetype runs a Unity parallel for job to calculate their vision, where each index is a separate unit of that archetype. Each unit's vision is calculated by using the archetype's edge deltas, the edge of the vision in relation to the unit, and using DDA to draw a line to each edge from the unit's starting point. The edge deltas are calculated on archetype creation and are found using a simple algorithm using pythagoras on one octant and then x and y values are shifted around for the other 7 octants. See pseudocode in Figure 10. It is also worth noting here that while this algorithm was "invented", so to speak, for the project, there are other algorithms that do this too. One of those is Brasenham's own circle drawing algorithm, which works in a similar way [29]. It is also possible to hardcode these edge deltas into the program per view distance, but that was not done in this project.

```
StaticXY (viewDistance, tolerance)
    int[] edgeArrayX, edgeArrayY
    float viewDistSq = (viewDistance + tolerance) *
                       (viewDistance + tolerance)
    int dy = viewDistance
    int dx, index

    //find up->right oct
    while (dy >= dx)
        if (dx*dx + dy*dy <= viewDistSq) //Pythagoras
            edgeArrayX[index] = dx
            edgeArrayY[index] = dy
            dx++
            index++
        else
            dy--

   int octantLen = index
   int[] newEdgeArrayX = new int[octantLen * 8]
   int[] newEdgeArrayY = new int[octantLen * 8]

   for (int i = 0; i < octantLen; i++)
      newEdgeArrayX[i] = edgeArrayX[i]
      newEdgeArrayY[i] = edgeArrayY[i]

   //right->up oct with x&y shifting
   for (int i = 0; i < octantLen; i++)
        dx = newEdgeArrayX[i]
        dy = newEdgeArrayY[i]
        newEdgeArrayX[index] = dy
        newEdgeArrayY[index] = dx
        index++

    /* Similarly for other octants (not shown)
     * All octants values are:
     * (nx, ny = new values, x, y = original)
     * up->right:   (x,y)
     * right->up:   nx =  y, ny =  x
     * up->left:    nx = -x, ny =  y
     * left->up:    nx = -y, ny =  x
     * left->down:  nx = -y, ny = -x
     * right->down:nx =  y, ny = -x
     * down->left:  nx = -x, ny = -y
     * down->right:nx =  x, ny = -y
     */
    return newEdgeArrayX, newEdgeArrayY
```

Figure 10: Pseudocode to calculate the edge cells of a unit's vision.

To ensure that there are no missed cells within the units vision, the DDA algorithm is not only performed on the outermost layer of cells, but also the next outermost too. This is done by decrementing the edge coordinate and running another DDA on that. The decrementing works by decrementing either x or y depending on whether dx or dy is bigger respectively. This makes it so the coordinate is decremented the way that takes it the closest to the start point. The line stops if it hits a wall or resource and the maximum amount of wall/resource penetration is reached. See Figure 11 for pseudocode of the full DDA algorithm used.

With the results back, the manager then sets the vision of each unit in the update list to the newly calculated ones. At the same time, units save their previous vision. Then all the units in the update list, and units whom vision intercepts or overlaps the same spatial squares as the units in the update list, are added to a new list which will be used to set the Color32 array. For each unit in this new list, the Color32 array is set to black for each index in their previous vision. After this reset stage, the Color32 array is set to white for each index in the units' new vision.

Finally a square texture is created with the world height and width, and its pixels are set with the Color32 array. The texture is then applied to the plane overlapping the game area, but invisible to the player. See Figure 12. This plane is black and white, and is used to mask another plane, using a shader and a Unity camera, which is the actual visual representation of the FOW.

In the solution, spatial indexing is employed to optimize iteration over a large number of units and cells. There are two main methods used for this. The first one is to limit which units need to be taken into account when updating the FOW. This is done by overlaying the world with a grid of square Unity colliders and attaching a sphere collider, with radius equal to the view distance, to each unit. These colliders are put in the same physics layer, which can only interact with itself. Then, when a unit is within range of a given square, their colliders will collide and pick up the collision event. It is through this event that the squares add the collided unit in their unit list, and the unit adds the square to their squares list. Then, when spatial indexing, each square in every unit that needs to update their FOW is accessed and all units within those squares are used when updating the FOW. This ensures that the FOW is only updated within a square of influence around the unit whom needs updates. This is an example of using the fixed grid index method.

The second way is to limit the amount of array indexing when we only need to index around a given location. This is, among others, used when copying wall/resource placements into a NativeArray for the Unity parallel for job. To minimize indexing, we create an MBR around the unit given its view distance and start index in the array. From the start index we can create start x- and y-values for the unit in the grid the array represents. Then, by using the start values, we can determine minimum and maximum x- and y-values given the unit's view distance. These form the MBR, which is subsequently used to restrict the copy area as seen in Figure 13.

The solution's performance is measured in the Unity Profiler, as written about in Appendix C. In the example given in Figure 14, a worst-case scenario test is performed which can be seen as the middle part where the average frames-per-second (fps) is between 100 and 60. Given that the fps is close to 150 when the game is running without any FOW updates, it means that in the worst case scenario the solution results in a ca. 100 fps drop.

```
FullDDA (x0, y0, dx, dy, indexArray)
    int x1 = x0 + dx //Precalculated edge deltas
    int y1 = y0 + dy

    int seenThroughCount
    float x = x0
    float y = y0
    if (dx == 0) //straight up or down
        while (y != y1)
            y += Sign(dy)
            int newIndex = MakeIndex
            indexArray[newIndex] = 1
            if (!ContinueLine(newIndex, ref seenThroughCount))
                break
    else
        float a = 0
        if(dx != 0)
            a = Abs(dy/dx)

        if (a > 1) //steep slope
            float rev = 1 / a
            while (y != y1)
                x += rev * Sign(dx)
                y += Sign(dy)
                int newIndex = MakeIndex
                indexArray[newIndex] = 1
                if (!ContinueLine(newIndex, ref seenThroughCount))
                    break
        else //a<=1, slow slope
            while (x != x1)
                x += Sign(dx)
                y += a * Sign(dy)
                int newIndex = MakeIndex
                indexArray[newIndex] = 1
                if (!ContinueLine(newIndex, ref seenThroughCount))
                    break

    //Notes:
    //seenThroughCount is a reference value.
    //ContinueLine method checks if the line should continue
    //based on if it hits an obstacle and the
    //seenThroughCount versus a maximum.
    //It increments seenThroughCount if it is lower than
    //maximum.
    //Sign(value) gives 1 or -1 based on whether or not the
    //value is positive or negative.
    //Abs(value) gives the absolute of the value.
```

Figure 11: Pseudocode which details the algorithm of DDA used in the final solution.

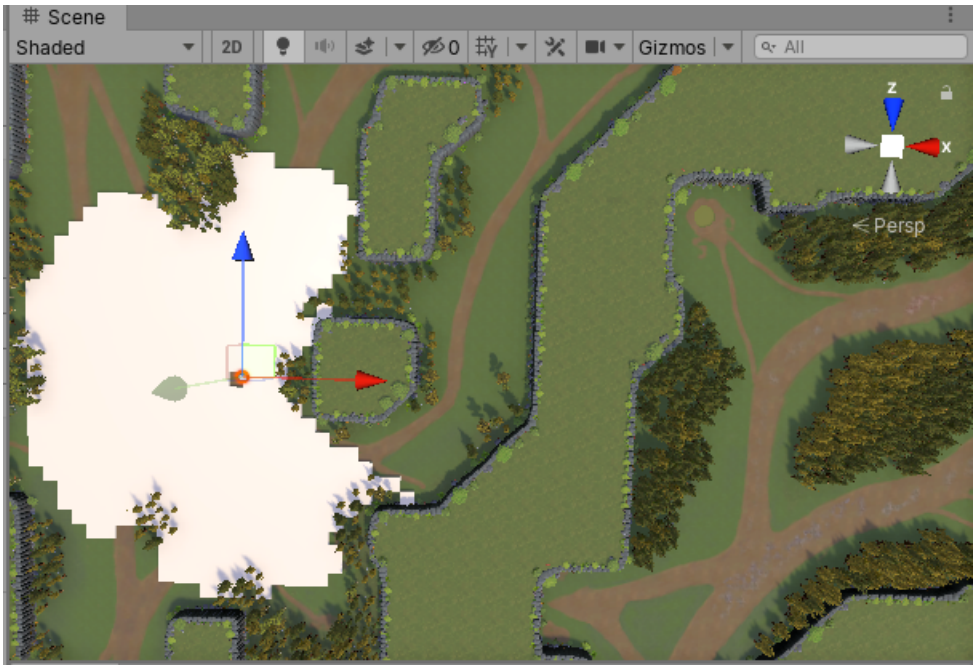


Figure 12: An image of the solution's vision plane, which would be invisible to the player. It shows how a unit's vision interacts with the environment with Line of Sight.

4.2 Engineering

In the Vision Document, Appendix D, required and optional functional properties of the solution is listed under 5.1 and 5.2. Also note these are directly linked to the user stories in Appendix B, the Requirements Document.

4.2.1 Crucial functional properties

Property	Status in solution
Lighting up area around friendly units	This works as intended.
Blocking view through terrain	Vision or view is blocked by terrain in the solution. This works well.
Blocking view through buildings	This works well too and is incorporated into the same system as the previous point.
Only being able to see X cells into the rocks in the mine	This is in and implemented by counting up towards a maximum limit each time the algorithm hits a resource/wall.
Performance	The solution's performance is good enough on its own when tested in worst case scenarios.

Table 3: Required functional properties of the solution.

```
SpatialIndex (startIndex, viewDistance, worldWidth)
    int x0 = startIndex % worldWidth
    int y0 = startIndex / worldWidth
    int minX = x0 - viewDistance
    int minY = y0 - viewDistance
    int maxX = x0 + viewDistance
    int maxY = y0 + viewDistance

    for (int x1 = minX; x1 < maxX; x1++)
        for (int y1 = minY; y1 < maxY; y1++)
            int index = y1 * worldWidth + x1
            CopyValue(index)
```

Figure 13: Pseudocode of the Spatial Indexing method MRB as used in the solution when copying spatial data.

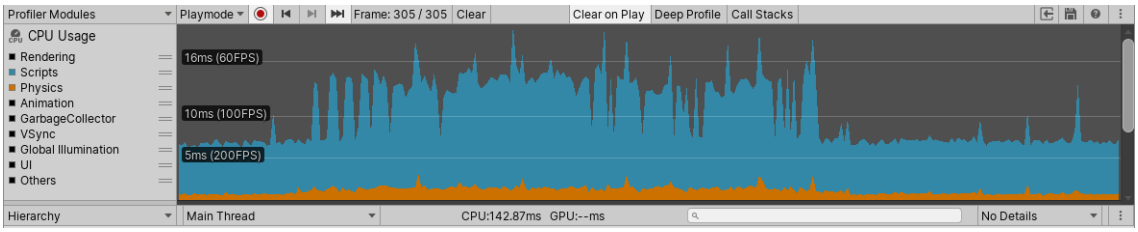


Figure 14: An example of the performance as viewed in the Unity Profiler.

4.2.2 Optional functional properties

Property	Status in solution
Grayscale of static objects map	Not implemented.
Grayscale of explored map	Not implemented, but the way the Color32 FOW array is set could be used to implement this.
Height based Line of Sight	Not implemented.
High ground sight	Not implemented.
Delayed vision loss	Not implemented.

Table 4: Optional functional properties of the solution.

4.3 Administrative

4.3.1 Project Plan

The project followed its plan in general, with only a handful changes. The requirements document was finished a week early, leaving more time for the logic development. However, the most major deviation from the plan was the almost complete lack of visual development, which gave room to more logic development and documentation writing. The system document was also written a week early, which in turn gave more time for the main report. This can be seen visualised in Figure 15.

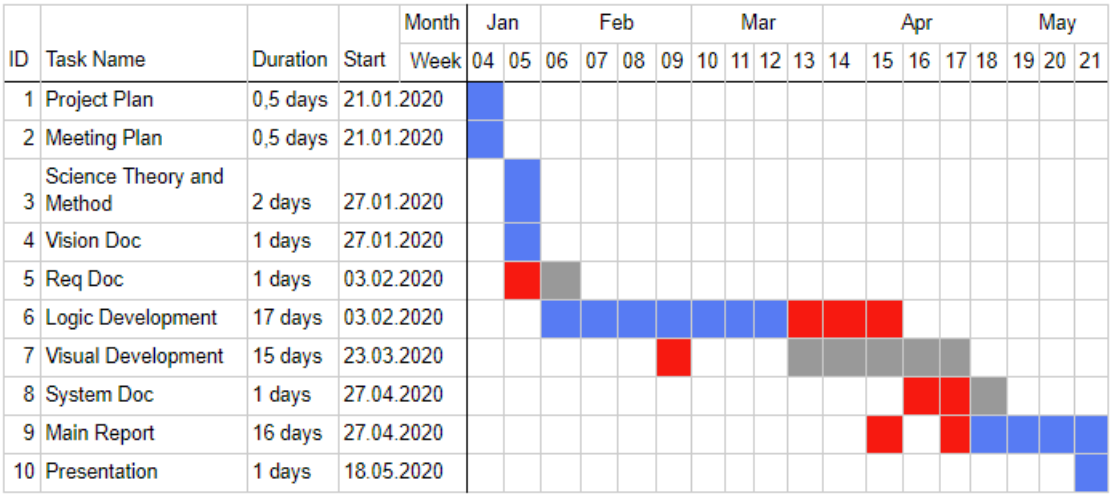


Figure 15: Gantt chart like the one in Appendix E, Project Plan. Blue is where the original plan and the actual execution lines up, red is the deviation in the actual execution and gray is where the plan was not executed. "Duration" and "Start" displays the planned duration of the activity and its planned start date.

4.3.2 Time Management

In general, most of the project's time was used on developing the solution and writing various documentation, including the main report. It has also been invested a fair amount of time into research of various relevant topics. See Appendix F, Timesheet for more thorough information on time management during the project.

Activity	Amount of time (hours)
Documentation	233,5
Programming	144,5
Research	58
Meetings	8
Emails	5,5
Other	10
Total	**459,5**

Table 5: Time used in the project.

5 Discussion

5.1 Scientific

The end product is a LOS based FOW with significant optimization, built primarily for the RTS genre. Even if RTS is its birthplace, it should work well for any game that has a grid-based world. The solution can be understood as an answer to the research problem, as a recipe for an optimized LOS based FOW. While the solution is created specifically in Unity and taking advantage of

Unity's packages and architecture, the overarching structure and how the units' vision is calculated remains universal.

Functionally, the solution works very well. It calculates each unit's vision perfectly in regards to their LOS and stitches all the visions together into the reverse of the FOW. The unit vision algorithm, using DDA to draw lines to edge cells, has been tested to work on multiple different view distances between 2 and 25. Multiple optimization methods are in use, including spatial indexing, parallelization and Unity Burst. These serve to speed up the solution. The employer was also impressed by the results, though there remains some work to incorporate it into the larger game.

One of the choices made were to use DDA over Bresenham. This was, as said, due to DDA being easier to implement at the time. However, Bresenham should theoretically have less impact on performance. This might not be very much in modern computers, but it might make an impact when you take into consideration the pure amount of line drawings necessary. Let us consider the worst case scenarios the solution was tested in. Calculating the vision of 120 units, each with a maximum view distance of 15. This means that the circumference of their vision is $C = 2\pi r = 2 * \pi * 15 = 94.25$, that is to say a maximum of 95 cells. That means the line drawing algorithm has to run a maximum of $95 * 2 = 190$ times per unit, as per edge cell the DDA algorithm is run twice. Take that times 120 units is equal to a maximum of 22 800 times per update. This is a high number of operations needed per FOW update, which is why using Bresenham over DDA might give larger benefit than expected.

Still, that is not taking in to account that calculating the units' visions is one of the *least performance heavy operations when creating the FOW*. This is, for better and worse, due to the implementation with Unity Jobs and the usage of the Burst Compiler. The Burst Compiler heavily optimizes the parallelized job, which makes it insanely fast. This lessens the potential benefit of implementing Bresenham over DDA in the first place.

The Unity Jobs and Burst implementation does have its drawbacks though. Unity Jobs are, in general, very restrictive. This restrictiveness leads to a lot of overhead when copying lists and arrays into NativeArrays and when untangling the results afterwards. However, the overhead of copying is somewhat offset by spatial indexing the arrays based on unit start position and view distance. In this case, the pros of parallelized jobs and Burst outweigh their drawbacks, but being able to further diminish the drawbacks would certainly improve the performance. One way to potentially save some overhead is by using NativeArrays consistently throughout the solution instead of regular arrays, though that might backfire if NativeArray is generally slower than a regular array.

Lastly, there is one method not explored in this project which might solve the performance problems of the final solution. That is the method of pre-baking the LOS. This is, similar to how the unit edge cells deltas are handled, the idea of preemptively calculating the LOS once and storing it. This could be done by hardcoding or calculating at game start each cell's LOS for given view distances. Then when a unit's LOS needs to be updated, it just copies the LOS for the correct cell it is in. This would eliminate the need to calculate the LOS using algorithms during runtime, which would increase the solution's performance. This method does have some drawbacks as well though. It requires the game world to be known before loading, which means it would not work as

well for randomly generated worlds. However, one could do the calculations during the loading of the world, assuming the world is not generated during runtime. In RTS games though, the world is almost always made beforehand. Therefore, it would seem it could lend itself well for RTS games. However, during runtime in most RTS games, resources are depleted and buildings raised. This would change the LOS of nearby tiles. Which means a solution would have to recalculate nearby tile's LOS during runtime, which reduces some of the initial advantages of the method.

5.2 Engineering

In terms of the requirements, all the crucial requirements were filled. The only point that might need reevaluation is the performance. While the solution performs well in worst-case scenarios, that is only on its own. The solution has not been tested in tandem with the larger game during a game match. There is the possibility that during gameplay the performance of the larger game itself is worse, which in turn would mean that adding the new FOW solution could be detrimental to the game's performance and framerate. Even so, during normal gameplay, it is unlikely that 120 units needs their vision updated at once. Good documentation of the testing and performance could also be a point of contention, as that is not available for this project.

None of the optional requirements were added, though the solution has revealed a way the "Grayscale of explored map" could be implemented. When setting the units' vision plane, white is their vision and black is the FOW. Incidentally though, the plane starts out as transparent. Which means that since the plane is set black only after a unit has already been there, the black areas act as the "expored, but not currently visible" map. Using this trio of colors, transparent as unexplored, black as explored, but not visible and white as visible, it should be possible to implement a grayscale of the explored areas that are currently not in your units' vision. Which colors are in use is arbitrary, as long as it is 3 separate ones.

In Appendix B, Requirements document, one user story mentions a "wiki". Internally, the employer uses wikis to document certain classes and directories. The idea was that as part of the project, such a wiki was to be made so that developers could familiarise themselves with the code. However, the wiki was not written in favour of Appendix C, System Documentation. It does much the same, including detailed architecture, class descriptions and API.

5.3 Administrative

The development process itself was not very set in stone. It was a free flowing approach to developing a solution, with both positives and negatives. In the end it worked mainly because the team consisted of one person, which could have a plan and tasks assigned without physically writing them down anywhere. This gave way to rapid development and innovation though. The solution was iterated on constantly and new technologies implemented. However, this has a major drawback as well. Documenting such a development process is next to impossible. In addition, there are no documented test results either. Being able to document the development process and test results is part for the course of any report. With this in mind I would suggest putting more effort into planning how to do this early on in any similar projects.

Even with a free flowing process, there was an overarching plan. Though there were a few points where the project deviated from this plan. This was mainly the drastic shortening of the visual development, and subsequent extension of logical development and main report writing. This was due to being able to use parts of the employer's previous implementation of the FOW for this purpose. The employer's previous solution was a simple white circle around each unit, which, by using a camera and a shader, masked a plane which was the actual FOW. By using the same camera, shader and actual FOW plane, the only thing needed to do was setting a plane white according to unit's vision.

Lastly, there was fewer meetings and less general involvement than recommended with the supervisor. Which means less feedback on various aspects of the project. This is partly due to the situation that found place during most of the project, which would be hard not to mention, the COVID-19 pandemic. While digital tools alleviated the impact of the social distancing, the inherent stress of the situation caused problems nonetheless. This was in no part helped by the fact that the supervisor had a habit of not responding to emails, even repeated ones. This resulted in general apathy from my side, preferring to just continue on with the project.

5.4 Own Effort and Learning

While COVID-19 was undoubtedly stressful, it did not affect my own effort too much. Digital tools such as Git and Discord helped bridge the digital gap, but the main reason I remained unaffected was due to the fact that the project was very self contained. This meant that working from home posed little threat to the project's course.

In addition, the project was quite interesting and educational to work on. I had no prior experience with line algorithms, which I found out is a broad field. While I only focused on DDA and Bresenham in this report, a few others were encountered as well. No doubt there are other, more modern line algorithms in widespread use as well.

I also had to learn about various forms of optimizations that could be applied to the solution, including spatial indexing, C# Parallel.ForEach, Unity Jobs and Burst Compiler. These were all invaluable for the final solution and the experience gained will no doubt be of use in the future.

All in all the project has been a great experience. I will not hide that most of the effort, at least in calendar time, went into developing a fit solution for the employer. As such, the surrounding documentation may feel a little ad-hoc. Even so, I am overall happy with the project and the effort I put into it.

5.5 Ethics

When working in any field you have to think about ethical concerns surrounding the project and its execution. Usually in software development most ethical problems arise from handling sensitive user data, which this project did not do. Even so, there are a few relevant ethical problems.

While the project would not include handling sensitive *user* data, it still entailed getting access

to the game's source code and plans. This information is private however, which means it should be handled with care. This was solved by the employer first issuing a Non-Disclosure Agreement (NDA). This legally binding document prohibits the disclosure of said sensitive data. Even without the document I would have handled the data with care and respect, but for the employer getting a written agreement is no doubt reassuring.

COVID-19 also becomes relevant once again. During such a pandemic it is crucial to work in a different manner than usual. It would not be ethical to work in an environment with multiple people, due to the risks of getting sick and spreading the disease. Getting sick can lead to spreading to strangers or loved ones. In severe cases, it might lead to one's own premature death. Early on it was therefore decided to work on the project from home. Not long after, the employer applied this to all employees.

Lastly, when developing the solution, I have made an effort to design the solution in an non-intrusive way. This means not editing their source code, but using it as expected when needed. This was done by making the solution as modular and self-contained as possible. In the final iteration it only depends on a few references from their source code.

6 Conclusion and Future Work

6.1 Conclusion

The final solution can be viewed as an answer to the research problem, as detailed in chapter 4.1. In short, one way to implement an optimized LOS based FOW solution in Unity is to draw lines using DDA between start and edge of vision cells on a per unit basis, then stitching the separate units' vision together, applying it as a black and white texture to a plane overlapping the game area, which then masks another plane which visualizes the FOW. To optimize this, use a central manager to batch together unit vision update on a timer, use spatial indexing to avoid unnecessary indexing of various arrays and use parallelization in the form of Parallel.ForEach and Unity Jobs with the Burst Compiler enabled.

6.2 Future Work

For future work, performance is what needs the most improvement. I imagine this could be improved through implementing Bresenham's line algorithm or perhaps structuring the whole implementation in a new way, maybe through data oriented design or pre-baking the LOS. It would also be interesting to implement LOS based FOW in other popular game engines, such as Unreal, Gamemaker and Godot.

6.3 Acknowledgements

While the problem to solve for this project was to create a FOW, there are no images of a final complete FOW. That is because, while this did work at a time, it encountered a shader error down

the line which I was unable to solve. This should be an easy fix for the employer though, who has the required knowledge of this.

Bibliography

[1] Pineleaf Studio. DwarfHeim; 2020. [Online; accessed 13-May-2020]. Available from: https://dwarfheim.com/.

[2] DeLoura MA. Game Programming Gems 2. Cengage Learning; 2001. [Online; accessed 13-May-2020]. Available from: https://books.google.no/books?id=1-NfBElV97IC.

[3] Moss R. Build, gather, brawl, repeat: The history of real-time strategy games. Ars Technica. 2017;[Online; accessed 13-May-2020]. Available from: https://arstechnica.com/gaming/2017/09/build-gather-brawl-repeat-the-history-of-real-time-strategy-games/.

[4] Buro M. Real-Time Strategy Games: A New AI Research Challenge; 2020. [Online; accessed 13-May-2020]. Available from: https://pdfs.semanticscholar.org/2c32/d813d58872c081a3ede56fe2439a2a30b793.pdf.

[5] Hsieh JL, Sun CT. Building a Player Strategy Model by Analyzing Replays of Real-Time Strategy Games. Taiwan; 2008. [Online; accessed 13-May-2020]. Available from: https://www.researchgate.net/profile/Ji-lung_Hsieh/publication/221533756_Building_a_Player_Strategy_Model_by_Analyzing_Replays_of_Real-Time_Strategy_Games/links/547342730cf2d67fc0360f1a.pdf.

[6] Micić A, Arnarsson D, Jónsson V. Developing Game AI for the Real-Time Strategy Game Starcraft. Reykjavik University. Iceland; 2011. [Online; accessed 13-May-2020]. Available from: https://pdfs.semanticscholar.org/05f9/07b964ea0fa4a3b12c411cd86d23da5d2b93.pdf.

[7] Peng P, Wen Y, Yang Y, Quan Y, Tang Z, Long H, et al.. Multiagent Bidirectionally-Coordinated Nets. England; 2017. [Online; accessed 13-May-2020]. Available from: https://arxiv.org/pdf/1703.10069.pdf.

[8] Blizzard Entertainment. StarCraft 2; 2020. [Online; accessed 13-May-2020]. Available from: https://starcraft2.com/en-us/media.

[9] Dahlskog S, Kamstrup A, Aarseth E. Mapping the game landscape: Locating genres using functional classification; 2009. [Online; accessed 13-May-2020]. Available from: https://www.researchgate.net/publication/236178139_Mapping_the_game_landscape_Locating_genres_using_functional_classification.

[10] von Clausewitz C. On War. English ed. Princeton University Press; 1976. [Online; accessed 13-May-2020]. Available from: http://slantchev.ucsd.edu/courses/ps143a/readings/Clausewitz%20-%20On%20War,%20Books%201%20and%208.pdf.

[11] Kiesling EC. On War Without the Fog. Military Review. 2001;[Online; accessed 13-May-2020]. Available from: http://www.clausewitz.com/bibl/Kiesling-OnFog.pdf.

[12] Heinz T. Location-based Game Design Pattern Exploration Through Agent-Based Simulation. Germany; 2017. [Online; accessed 13-May-2020]. Available from: https://www.researchgate.net/publication/236178139_Mapping_the_game_landscape_Locating_genres_using_functional_classification.

[13] Gregory J. Game Engine Architecture. 3rd ed. CRC Press; 2018. [Online; accessed 13-May-2020]. Available from: https://books.google.no/books?id=EwlpDwAAQBAJ.

[14] Gomes J, Velho L, Sousa MC. Computer Graphics: Theory and Practice. CRC Press; 2012. [Online; accessed 13-May-2020]. Available from: https://books.google.no/books?id=ID1tP9DfKgEC.

[15] McKinney AL. Development of the bresenham line algorithm for a first course in computer science. Consortium of Computer Sciences in Colleges. 1992;[Online; accessed 13-May-2020]. Available from: http://www.ccscjournal.willmitchell.info/Vol8-92/No4/Alfred%20L%20McKinney.pdf.

[16] Worboys MF. GIS: A Computer Science Perspective. CRC Press; 1995. [Online; accessed 13-May-2020]. Available from: https://books.google.no/books?id=duT2fcnQeJMC.

[17] Merriam-Webster. raster; 2020. [Online; accessed 13-May-2020]. Available from: https://www.merriam-webster.com/dictionary/raster.

[18] Bresenham JE. Algorithm for computer control of a digital plotter. IBM Systems Journal. 1965;4(1). [Online; accessed 13-May-2020]. Available from: https://web.archive.org/web/20080528040104/http://www.research.ibm.com/journal/sj/041/ibmsjIVRIC.pdf.

[19] Flanagan C. The Bresenham Line-Drawing Algorithm; 2020. Available from: https://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html".

[20] Zhang X, Du Z. Spatial Indexing; 2017. [Online; accessed 13-May-2020]. Available from: https://gistbok.ucgis.org/bok-topics/spatial-indexing.

[21] Borufka R. Performance Testing Suite for Unity DOTS. Charles University. Czech Republic; 2020. [Online; accessed 13-May-2020]. Available from: https://dspace.cuni.cz/bitstream/handle/20.500.11956/116800/120352992.pdf?sequence=1&isAllowed=y.

[22] Ramaswamy S. Efficient Field of View and Line of Sight for strategy games. Gamasutra. 2016;[Online; accessed 13-May-2020]. Available from: https://www.gamasutra.com/blogs/SundaramRamaswamy/20161117/285708/Efficient_Field_of_View_and_Line_of_Sight_for_strategy_games.php.

[23] Wu X. An Efficient Antialiasing Technique. Computer Graphics. 1991;25(4). [Online; accessed 13-May-2020]. Available from: https://dl.acm.org/doi/pdf/10.1145/127719.122734.

[24] Leler WJ. Human Vision, Anti-aliasing, and the Cheap 4000 Line Display. Computer Graphics. 1980;[Online; accessed 13-May-2020]. Available from: https://dl.acm.org/doi/pdf/10.1145/965105.807509.

[25] Microsoft. Thread Class; 2020. [Online; accessed 13-May-2020]. Available from: https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread?view=netcore-3.1.

[26] Microsoft. Parallel.For Method; 2020. [Online; accessed 13-May-2020]. Available from: https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.for?view=netcore-3.1.

[27] Microsoft. Parallel.ForEach Method; 2020. [Online; accessed 13-May-2020]. Available from: https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel.foreach?view=netcore-3.1.

[28] Unity. C# Job System; 2020. [Online; accessed 13-May-2020]. Available from: https://docs.unity3d.com/Manual/JobSystem.html.

[29] Cao M, Liu S, Cao F. Midpoint Distance Circle Generation Algorithm based on Midpoint Circle Algorithm and Bresenham Circle Algorithm. Journal of Physics: Conference Series. 2020;1438. [Online; accessed 13-May-2020]. Available from: https://iopscience.iop.org/article/10.1088/1742-6596/1438/1/012017/pdf.

Appendix

A Original Task Description

Arbeidstittel:	Spillutvikler for Pineleaf Studio

Hensikten med oppgaven:
Utvikle en bedre løsning for ''Fog of War'' for RTS-spill, med utgangspunkt i DwarfHeim.

Kort beskrivelse av oppgaveforslag:
Studenten skal utvikle løsninger på tekniske og designmessige utfordringer for ''Fog of War'' for RTS-spill. Ekstra utfordringer med oppgaven er blant annet ytelse, optimalisering, tydelighet og visuell opplevelse. Utviklingen skal hovedsaklig foregå i Unity3D med C#.

''Fog of War'' er kort forklart synsfeltet til spilleren basert på vennlige enheter og bygninger.

What is DwarfHeim? https://store.steampowered.com/app/977650/DwarfHeim/

Oppgaven passer for (kryss av de(t) som passer og skriv evt. en kommentar til oss):	- Bacheloroppgave - Sindre Haugland Paulshus <sindrhpa@stud.ntnu.no>
Kan oppgavestiller stille arbeidsplass med nødvendig utstyr og programvare:	Ja.
Begrensninger i tilgjenglighet av opplysninger o.l:	Kildekode til spillet kan ikke publiseres uten sensureres.
Oppgaven passer best for, antall studenter:	- 1
Opplysninger om oppgavestiller	
Navn på bedrift eller organisasjon:	Pineleaf Studio
Addresse	Munkegata 58
Postnummer	7011
Poststed	TRONDHEIM
Navn på kontaktperson/veileder:	Sigurd Murad
Telefon:	90027573
Epost:	sigurd.murad@pineleafstudio.com
Navn på kontaktperson 2/veileder 2:	Kenneth Kristensen
Telefon kontaktperson 2/veileder 2:	99644566
Epost kontaktperson 2/veileder 2:	kenneth.kristensen@pineleafstudio.com

Utfyllende kommentarer til hva oppgaven gjelder:
Oppgaven er utviklet i samarbeid med Sindre H. Paulshus knyttet til Bacheloroppgaven hans. Handler om å finne en god løsning på problemer knyttet rundt synsfelt i strategispill.

B Requirements Document

Requirements Document

"Fog of War"

Bachelor's Thesis 086
Sindre Haugland Paulshus

Date	Version	Description
21.01.2020	0.1	Initial setup.
03.02.2020	1.0	First and final draft.

Table of Contents

1. Introduction

This document details the specific requirements to be met by the solution for Fog of War (hereby "FOW") for the game Dwarfheim, being developed by Pineleaf. The solution will be worked on by a single bachelor student for his bachelor's thesis during the first semester of 2020.

2. User stories

2.1. The Player

As a	Player.
I want	To see the area around friendly units and buildings.
So that	I can make meaningful choices and execute on my strategy.
Acceptance Criteria	● If I am a player, I have vision with a certain radius around friendly units and buildings. ● I do not have vision around enemy units. ● If friendly units or buildings are destroyed, their vision will linger for a little while before fading out.

As a	Player.
I want	To have my vision be blocked by natural obstacles like trees, mountains, terrain.
So that	I can be immersed in the game, do special maneuvers and use the blocked vision of my enemies to my advantage.
Acceptance Criteria	● If I am a player, my vision is blocked by trees, mountains, higher terrain, walls and enemy buildings. ● In the mine, my vision will penetrate X layer(s) of stone.

2.2. The Developer

As a	Developer.
I want	To be able to use the FOW solution easily.
So that	I can implement FOW without having to use much time or effort.
Acceptance Criteria	● If I am a developer, I can skim over the source code and its comments to get an idea of how it works.

	● I can read the wiki to learn how to implement the FOW in the game.

As a	Developer.
I want	To be able to edit the FOW solution easily.
So that	I can edit parameters or change its behaviour if needed.
Acceptance Criteria	● If I am a developer, I can skim over the source code and its comments to get an idea of how it works. ● I can read the wiki to learn how all the various parts of the solution coexist and where to find specific parameters so that I can edit them.

C System Documentation

System Documentation

"Fog of War"

Bachelor's Thesis 086
Sindre Haugland Paulshus

Date	Version	Description
21.01.2020	0.1	Initial setup.
15.04.2020	0.2	Introduction added. Definitions and references added.
17.04.2020	0.3	Added class and architecture diagrams.
21.04.2020	1.0	First full draft. Added link to source code documentation, wrote about testing, edited the diagrams, added class explanations.

Table of Contents

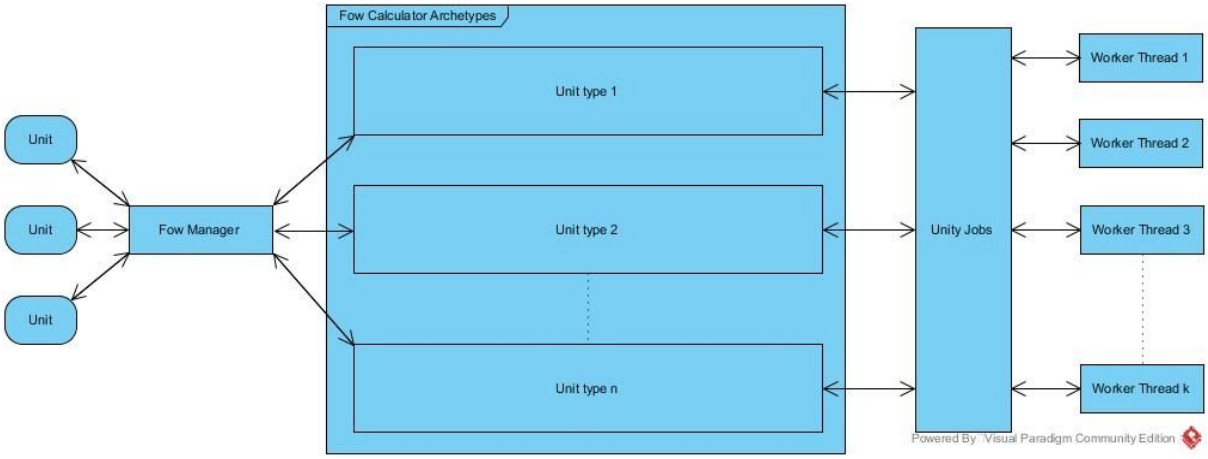
1. Introduction

This document describes the "Fog of War" (herby "FOW") solution created for the game Dwarfheim, being developed by Pineleaf Studio[1]. The project itself is staffed by a single bachelor student for his bachelor's thesis during the first semester of 2020. This document includes the overarching architecture of the solution, its classes, source code documentation and how testing were done.

1.1. Abbreviations, Acronyms and Definitions

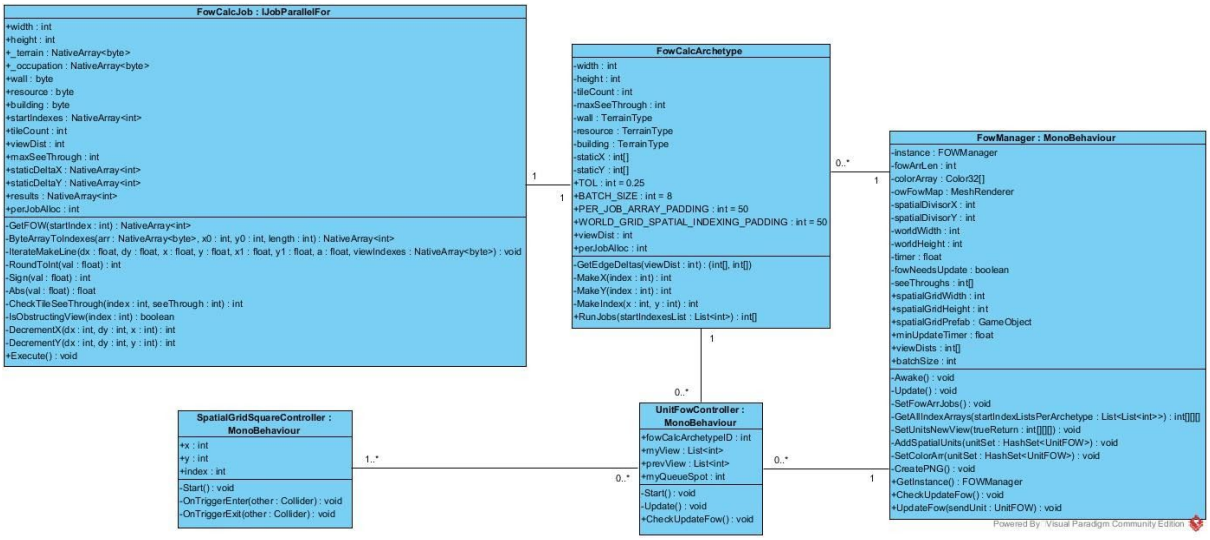
Prefab	A saved configuration of a GameObject. Can be instantiated in a scene.
FOW or Fow	Fog of War
Unity Profiler	A tool within Unity which records cpu and gpu usage, frame rate and various other data for the running game. It also shows which components, classes and functions are responsible for causing said data.

2. Architecture



The solution works in layers. Separate units communicate with the FowManager when their view needs to update. This happens when either the unit is spawned or the unit moves. The manager then requests the view for each unit that needs to update using Fow Calculator Archetypes and the units' spot (index) in the world grid. Each archetype schedules one Unity Parallel For Job, where each index it works on is a separate unit of that archetype. Therefore, each unit's view is calculated on a unique thread, which takes advantage of multiple CPU cores and boosts performance.

3. Class Diagram



The project consists of 5 classes:

SpatialGridSquareController	A component class of the SpatialGridSquare prefab. An amount of these prefabs are spawned in a square grid and help with spatial indexing the units in the game. Has two main functions, OnTriggerEnter() and OnTriggerExit(). They are called when collider enters and exits the collider attached to the SpatialGridSquare prefab respectively. It checks if the collider belongs to a unit and then adds or removes the unit from its list.
UnitFowController	This script is attached to the units in the game. Each frame it checks if itself has moved from the last frame. If it has, it tells the FowManager to update. Contains an ID int of which FowCalcArchetype it should use.
FowManager	A singleton class. Controls the flow of information for the fog of war. It checks on a configurable interval if any units have told it to update. It calculates the fow of the units that told it to update by separating them into archetypes and using the FowCalcArchetype class.
FowCalcArchetype	A unique set of a view distance and a maximum number of resources/walls it can see through. When created, it calculates the edges of this archetype's view in relation to the unit. Its main function is RunJobs, which initializes and schedules FowCalcJob.

FowCalcJob	Not actually a class, but a struct inside FowCalcArchetype. As such, it acts as a private inner class of the aforementioned. This is a Unity Parallel For Job, which means when scheduled, threads allocated by the Unity Jobs system will work on each index of an array separately using this struct. Each index of the array is references a unique unit of this archetype.

4. Documentation of Source Code

Source code documentation is written in markdown and available here: https://github.com/Sindrex/Bachelor086-fow-doc

5. Testing

Testing was done manually through the Unity Editor. There were two types of tests conducted during the development: Function testing and performance testing.

5.1. Function testing

Function testing is to check that the core functions of the system works, that the fog of war is created correctly. This is done by using two test units of two different archetype and moving one or both. The fog of war is observed as it changes depending on unit location and obstacles.

5.2. Performance testing

Performance testing aims to test the performance of the system in a worst case scenario. To do so, 120 test units (120 is the upper limit of units in the game) split into 4 archetypes are used and moved around simultaneously for a duration of 10-20 seconds. This forces the FowManager to update the fog of war and update the view of each unit on each check it does. During this, the Unity Profiler is recording the performance. Generally, a game frame rate of at least 60 fps is considered good, but this needs to be compared to the frame rate when the units are not moving to be able to see the impact of the fog of war. The observer can then identify which parts of the solution that is causing problems and set out to fix it.

6. References

[1] Pineleaf Studio. Dwarfheim [internet]. Pineleaf Studio; 07.02.2020 [updated 07.02.2020; 07.02.2020]. Available from: https://dwarfheim.com

D Vision Document

Vision Document

"Fog of War"

Bachelor's Thesis 086
Sindre Haugland Paulshus

Date	Version	Description
21.01.2020	0.1	Initial setup.
24.01.2020	0.2	First draft.
31.01.2020	0.3	Edited product properties after a meeting with the customer. Added "options for additions" as 5.2.
03.02.2020	0.4	Added a descriptive line in 3.5 alternatives.
07.02.2020	1.0	Tidied up the references with Vancouver style. Added a description to the 4.1 sketch. Final draft.

Table of Contents

1. Introduction

This document describes the overarching vision for the project. This includes overviews of the problem, product, stakeholders, users and requirements. In short, the project consists of making a "Fog of War" (herby "FOW") solution for the game Dwarfheim, which is being developed by Pineleaf Studio[1]. The FOW will be integrated into a larger project, the game itself. The project will be worked on by a single bachelor student for his bachelor's thesis during the first semester of 2020.

1.1. References

[1] Pineleaf Studio. Dwarfheim [internet]. Pineleaf Studio; 07.02.2020 [updated 07.02.2020; 07.02.2020]. Available from: https://dwarfheim.com
[2] Unity Technology. Unity [internet]. Unity Technology; 07.02.2020 [updated 07.02.2020; 07.02.2020]. Available from: https://unity.com/
[3] Max Proude. Ultimate Fog of War [internet]. Max Proude; 02.12.2016 [updated 20.06.2018; 07.02.2020]. Available from: https://assetstore.unity.com/packages/tools/utilities/ultimate-fog-of-war-76011
[4] Hukha. Lumbra [internet]. Hukha; 02.10.2018 [updated 07.01.2019; 07.02.2020]. Available from: https://assetstore.unity.com/packages/tools/particles-effects/lumbra-2d-dynamic-lights-and-field-of-view-128759
[5] Brendan L.K. Fog Of War [internet]. Brendan L.K. 17.12.2015 [updated 09.12.2019; 07.02.2020]. Available from: https://assetstore.unity.com/packages/tools/particles-effects/fog-of-war-51344

2. Summary of Problem and Product

2.1. Problem Summary

Problem with	The FOW solution for the game Dwarfheim. It is a simple circle and does not feature line of sight.
Affects	The players who play the game, and game developers.
And as a result	Game immersion may be spoiled and developers have to design around the fact.
A successful solution will	Seem natural to the players and offer the developers new opportunities within the game's design.

2.2. Product Summary

For	Pineleaf Studio.
That	Wants a new FOW solution.
The new FOW	Is a better FOW solution.
That	Features line-of-sight, is modular and easily used by other developers.
In contrast to	the current FOW that is only a simple circle, and other solutions you can buy which are hard to integrate into the game.
Our product	Features line-of-sight and is created specifically for Dwarfheim, and as such integrated with the game from the start.

3. Description of Stakeholders and Users

3.1. Summary of Stakeholders

Name	Description	Role during development
Pineleaf Studio	The client. Represented by the Lead Programmer and the CTO. Their livelihood are dependent on the game's success.	Through representatives, Pineleaf Studio will guide the development and offer valuable feedback.
Developers	The other developers of the game. Their livelihood are dependent on the game's success.	May offer valuable feedback, knowledge and insight into various aspects of the game, design and programming.
Sindre H. Paulshus	The creator of the FOW solution. Dependent on the solution's success for his grade.	Will be developing the solution.

3.2. Summary of Users

Name	Description	Role in development	Represented by
Players	The ones that play the game when it is finished. Wants the game to be as good as possible.	May offer feedback on the product during development.	Themselves.
Developers	Developers of the game. Will be using, maintaining and possibly editing the source code after the product is finished.	Same as in 3.1.	Themselves.

3.3. User Environment

The solution must fit into their existing architecture for the game, using Unity [2] and C# to create it. It must fit into the game's aesthetic and work seamlessly. Source code will be reviewed and distributed using Git. The solution should be modular and not affect code of the larger project. It should also be well documented and tidy, so it can be used, modified and maintained by other developers after its completion.

3.4. Summary of Users' Needs

Need	Priority	Affects	Current Solution	Recommended Solution
Being able to see within a radius of friendly troops and buildings	High	Gameplay	Shader and a simple circle that unmasks the map	Shader and dynamic area object that unmasks the map
Line-of-sight: Terrain. Not being able to see through trees, walls etc.	High	Gameplay	None	Dynamic area object that stops if it meets terrain

Line-of-sight: Buildings. Not being able to see through enemy buildings	Medium	Gameplay	None	Dynamic area object that stops if it meets buildings
Visual representation of the Fog of War	Medium	Gameplay	A shown area and fadeout at borders	Same as current.
Grayscale of explored map	Low	Gameplay	Only of the static game pieces.	Grayscale map of the explored map with static and dynamic game pieces (enemies where last seen).

3.5. Alternatives to Our Product

A common thread between all alternatives is that it is a generic package solution, and would not be tailor-made for the game. As such, it could be hard to make it work as you want or expand or edit it if needed.

Alternatives	Description
Ultimate Fog of War [3]	A FOW solution sold privately on the Unity Asset Store.
Lumbra [4]	A FOW solution sold privately on the Unity Asset Store.
Fog Of War [5]	A FOW solution sold privately on the Unity Asset Store.

4. Product Overview

4.1. Role in the User Environment

The product will limit the visibility and information available to the players. It will play a role in how the players interact with, view and play the game. See sketch below for how it is imagined the product will work.

The sketch shows the player's view in white and the fog in dark gray. Note that the view is actually a circle with a given radius. The middle icon is a player's unit and the mountain icons are terrain the unit cannot see through. As such, the terrain casts a shadow which limits the player's view.

4.2. Dependencies

Dependency	Description
Dwarfheim	The solution will be created specifically for Dwarfheim and as such be dependent on it to function.
Unity	The solution will be created in Unity for a Unity game. It can only be used with Unity.

5. Functional Properties of the Product

5.1. Crucial Properties

These are the main and most crucial properties of the product. These are expected to be fulfilled for the product to be finished.

Property	Description
Lighting up area around friendly	As a part of the Fog of War, areas

units	around friendly units will have to light up to give the player sight.
Blocking view through terrain	The player's sight will be limited by terrain of higher height (walls, cliffs, mountains, trees, etc).
Blocking view through buildings	The player's sight will be limited by buildings, especially enemy buildings.
Only being able to see X tiles into the rocks in the mine	The game has a mine world with its own map and there is squared stone and ore as walls. The player should initially only be able to see the outer layer of the rock. This should be customizable, so for example after an upgrade they might be able to see 2 or 3 tiles.
Performance	The product should have little or no impact on the game's performance (frames per second).

5.2. Options for Additions

In the case of all the main properties being finished with ample time left, these are options for additions that would enhance the product further.

Property	Description
Grayscale of static objects map	Areas the player has no sight in will be grayscale, but still show static objects such as terrain, trees and rocks.
Grayscale of explored map	Areas the player has no sight in will be grayscale. If the player has been there at one point during the gameplay and there was enemy buildings there, those will remain in grayscale. Note that the grayscale image will not update even if the building changes. To update the player has to explore it again.
Height based Line of Sight	The player cannot see playable areas of higher height than their own. Contrary, the player can see

	playable areas of lower height than their own.
High ground sight	The player's sight is extended when viewing lower areas than their own.
Delayed vision loss	Upon losing vision (ie. a unit or building is destroyed), the vision will linger for a time before fading out.

6. Non-functional Properties and other Requirements

6.1. Documentation

- Vision document
- Requirements Document
- System Documentation
  - Wiki
  - Commented source code

6.2. Tools

- The development of the solution must happen in Unity 2019.3.0f3 with C#.
- Code and file distribution will be done with Git and Azure devops.
- Coding will be done with Visual Studio Code.
- Visual Studio Code will need the extensions "C# XML Documentation Comments" and "Azure Repos".

6.3. Other

- Aesthetically pleasing visual representation.
- Submission of Main Report and all linked documentation.

E Project Plan

Project Plan

"Fog of War"

Bachelor's Thesis 086
Sindre Haugland Paulshus

Date	Version	Description
21.01.2020	0.1	Initial draft.
24.01.2020	1.0	Removed the prestudy, as client did not want a prestudy to be done. Edited gantt chart to reflect the change. Added system doc to milestones.

1. Project Description

I am tasked with developing a "Fog of War" (hereby "FOW") solution for the game Dwarfheim, being developed by Pineleaf Studio. The solution is supposed to be an upgrade from their current one and feature line-of-sight for certain objects and terrain. It also has to be visually appealing and have a low or non-existent impact on the performance of the game. It is preferred if the solution is modular as to not impact code for other parts of the game and be easily maintainable.

2. Goals

Effect goals:

- Create a good user experience for the game when it comes to FOW.
- Have users play the game for longer.
- Get more users to play the game.

Process goals:

- Get an understanding of how FOW can be developed.
- Get an understanding of how Unity shaders work and can be used to develop FOW.
- Get insight into game development.

Result goals:

- Create a FOW solution that features terrain/height blocking, line of sight and has a very low impact on performance.
- Create a FOW solution that is modular, easily used by other programmers and has no impact on already developed code for the game.

3. Milestones

Milestones for the project:

- Planning done.
- Science Theory and Method course completed.
- Vision document and requirements document done.
- Logical backbone of the FOW done.
- Visual representation of the FOW done.
- System documentation done.
- Main Report done.
- Presentation done.

4. Gantt Chart

ID	Task Name	Duration	Start	Month	Jan		Feb				Mar				Apr					May		
				Week	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21
1	Project Plan	0,5 days	21.01.2020																			
2	Meeting Plan	0,5 days	21.01.2020																			
3	Science Theory and Method	2 days	27.01.2020																			
4	Vision Doc	1 days	27.01.2020																			
5	Req Doc	1 days	03.02.2020																			
6	Logic Development	17 days	03.02.2020																			
7	Visual Development	15 days	23.03.2020																			
8	System Doc	1 days	27.04.2020																			
9	Main Report	16 days	27.04.2020																			
10	Presentation	1 days	18.05.2020																			

5. Gantt Chart in text

ID	Task	Week(s)	Description
1	Project Plan	4	Break the whole project up in discrete parts. Set a plan for when to work on the different parts.
2	Meeting Plan	4	Make a plan for when to meet with the supervisor.
3	Science Theory and Method	5, 7	A few obligatory days to learn about scientific theory and the scientific method.
4	Vision Document	7, 8	A document outlining the overarching requirements of the project. Should include features, risk analysis, cost analysis and description of the users of the system.
5	Requirements Document	8	Functional requirements of the project. Contains at least user stories.
6	Logic Development	9, 10, 11, 12, 13	Development of the underlying logic of FOW. Iteration based.
7	Visual	13, 14, 15, 16,	Development of the visual

	Development	17	representation of the underlying logic of FOW. Iteration based.
8	System Document	18	A document outlining the system, its architecture, class diagrams etc.
9	Main Report	18, 19, 20, 21	Write and finalize the main report.
10	Presentation	21	Create and finalize the presentation that is to be held of the project.

6. Quality Assurance

6.1. Testing

Code and functions developed will be continuously tested using Unity's built in features to play the game and Unity's console to print interesting data. Playing the game will act as integration testing, as it will show how the new features act in tandem with other parts of the game.

6.2. Code Review

As part of the game development of Dwarfheim, any finished code will have to get reviewed by two other staff members (peer review), one being the lead programmer, before it is allowed onto the development branch of the game. This will secure code quality and functionality.

7. Timesheet

As part of the project, a timesheet is to be used. It will be made using Google Sheets.

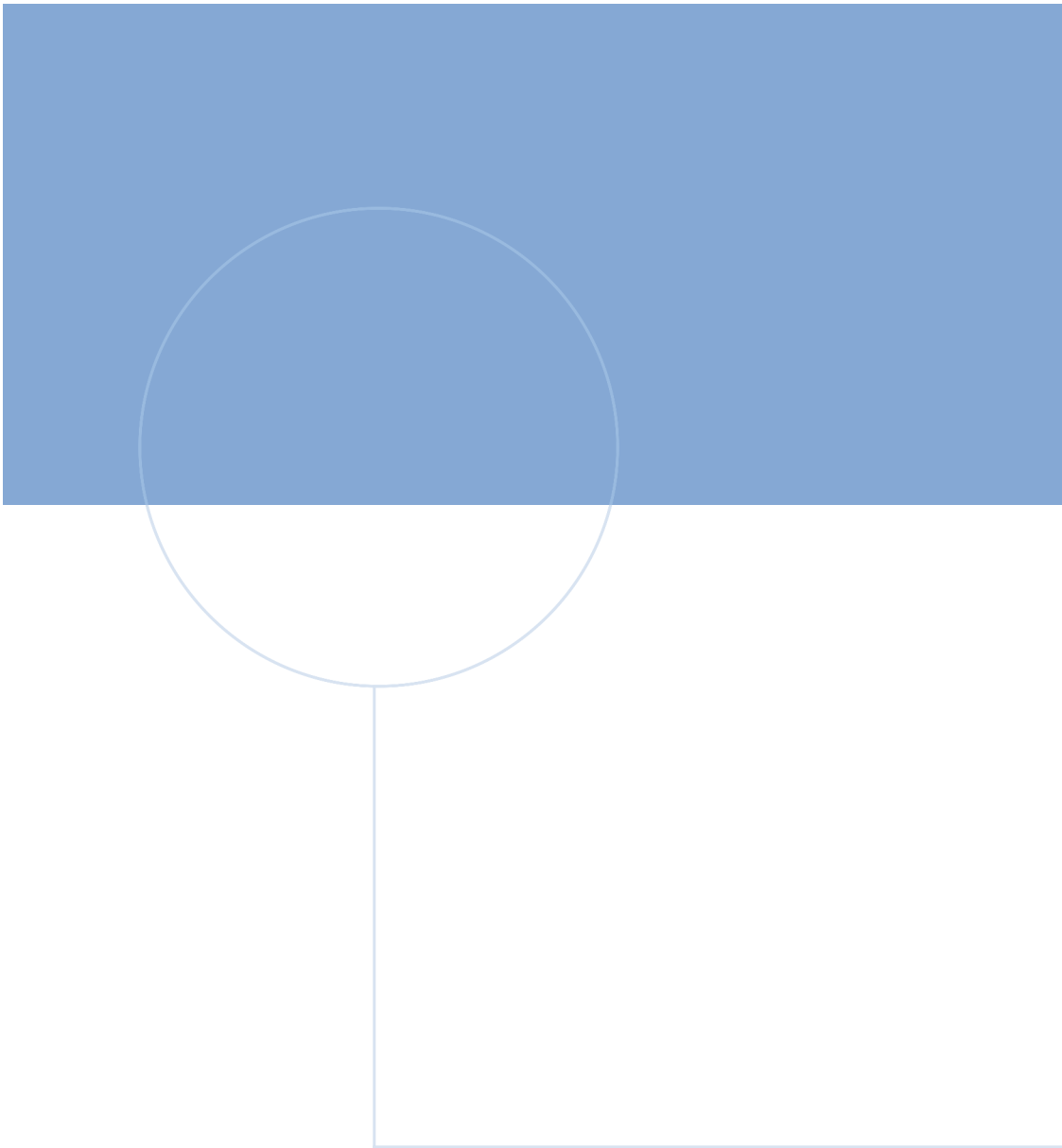
Link to the timesheet:
https://docs.google.com/spreadsheets/d/1JEzQuY3GiJ9b8ucb5ttH1dcnu6xxqOKKVaW12PprQWY/edit?usp=sharing

F Timesheet

Date	Documentation	Programming	Meetings	Emails	Research	Other	Note	Total day	Total Week	Total acc	Total percent	Planned acc	Dif
10.01.2020				1			Emails	1	1	1	0,2	3,8	−2,8
13.01.2020	2						Google Drive setup	2	3	3	0,6	7,7	−4,7
14.01.2020	1			1			Emails + timesheet setup	2	5	5	1	11,5	−6,5
15.01.2020								0	5	5	1	15,4	−10,4
16.01.2020	0,5						Meeting summons	0,5	5,5	5,5	1,1	19,2	−13,7
17.01.2020	2		2		2	2	Startup with Pineleaf, startup meeting w/ supervisor, meeting report, translation to english, setup git and repositories, research while waiting	8	13,5	13,5	2,7	23,1	−9,6
18.01.2020								0	13,5	13,5	2,7	26,9	−13,4
19.01.2020								0	13,5	13,5	2,7	30,8	−17,3
20.01.2020								0	0	13,5	2,7	34,6	−21,1
21.01.2020	5				1	1	9-16, finding out what needs to be done (docs), gantt chart, project plan, meeting plan, setup other docs	7	7	20,5	4,1	38,5	−18,0
22.01.2020								0	7	20,5	4,1	42,3	−21,8
23.01.2020								0	7	20,5	4,1	46,2	−25,7
24.01.2020	7						Revisions to project plan, finished first draft vision document, weekly report, meeting summons 2	7	14	27,5	5,5	50,0	−22,5
25.01.2020								0	14	27,5	5,5	53,8	−26,3
26.01.2020								0	14	27,5	5,5	57,7	−30,2
27.01.2020					5		Science day	5	5	32,5	6,5	61,5	−29,0
28.01.2020							Sick	0	5	32,5	6,5	65,4	−32,9
29.01.2020					3		Science day	3	8	35,5	7,1	69,2	−33,7
30.01.2020						1	Workshop presentation	1	9	36,5	7,3	73,1	−36,6
31.01.2020	1,5		0,5			2	Workshop 9:30-11:15, work 12-14, Requirements meeting, edited vision doc	4	13	40,5	8,1	76,9	−36,4
01.02.2020								0	13	40,5	8,1	80,8	−40,3
02.02.2020								0	13	40,5	8,1	84,6	−44,1
03.02.2020	2	5					Weekly report, meeting summons, req doc, experimentation (calculation based)	7	7	47,5	9,5	88,5	−41,0
04.02.2020		5					Experimentation (raycasting, Bresenham lines)	5	12	52,5	10,5	92,3	−39,8
05.02.2020			1			1	Meeting supervisor (whom did not show up), setup git repo for experiment	2	14	54,5	10,9	96,2	−41,7
06.02.2020		2	1				Programming, meeting supervisor	3	17	57,5	11,5	100,0	−42,5
07.02.2020	1	6			1		Experimentation (DDA + DDAMine, circle edges), edited vision doc, demo	8	25	65,5	13,1	103,8	−38,3
08.02.2020								0	25	65,5	13,1	107,7	−42,2
09.02.2020								0	25	65,5	13,1	111,5	−46,0
10.02.2020		0,5					innovation camp, 8:30-16	0,5	0,5	66	13,2	115,4	−49,4
11.02.2020							innovation camp 8:30-16	0	0,5	66	13,2	119,2	−53,2
12.02.2020							innovation camp 9-15	0	0,5	66	13,2	123,1	−57,1
13.02.2020								0	0,5	66	13,2	126,9	−60,9
14.02.2020	1,5	5		0,5			9-16, meeting summons 3, weekly report 4, circle edge work (2 thick) and grid edge fixes	7	7,5	73	14,6	130,8	−57,8
15.02.2020								0	7,5	73	14,6	134,6	−61,6
16.02.2020								0	7,5	73	14,6	138,5	−65,5
17.02.2020		5					9-14, tiles array -> bit/view arrays, experiment with Dwarfheim integration	5	5	78	15,6	142,3	−64,3
18.02.2020	0,5	6,5					9-16, solution into Dwarfheim, visualising solution in Dwarfheim, weekly report 5	7	12	85	17	146,2	−61,2
19.02.2020			1,5				meeting supervisor	1,5	13,5	86,5	17,3	150,0	−63,5
20.02.2020						2	work home, get repository on desktop pc	2	15,5	88,5	17,7	153,8	−65,3
21.02.2020		7					9-16, edge tile calculation optimization using octants	7	22,5	95,5	19,1	157,7	−62,2
22.02.2020								0	22,5	95,5	19,1	161,5	−66,0
23.02.2020								0	22,5	95,5	19,1	165,4	−69,9
24.02.2020	1	4					9-14, weekly report 6, iterating on solution in Dwarfheim using Singleton & unit scripts	5	5	100,5	20,1	169,2	−68,7
25.02.2020		5					work home 10-15, optimalization of delete/spawn visualiser (how to place fow efficiently)	5	10	105,5	21,1	173,1	−67,6
26.02.2020		5,5					work home 9:30-15, bitarray to plane material/image (png/color) with transparancy	5,5	15,5	111	22,2	176,9	−65,9
27.02.2020		1					work home: implement prev in Dwarfheim / CCD prep	1	16,5	112	22,4	180,8	−68,8
28.02.2020							CCD	0	16,5	112	22,4	184,6	−72,6
29.02.2020								0	16,5	112	22,4	188,5	−76,5
01.03.2020								0	16,5	112	22,4	192,3	−80,3
02.03.2020	0,5	5,5					9-15, weekly report 7, improve new fow visualizer, optimalize fow refresh	6	6	118	23,6	196,2	−78,2

Date	Documentation	Programming	Meetings	Emails	Research	Other	Note	Total day	Total Week	Total acc	Total percent	Planned acc	Dif
03.03.2020		3					10:30-13:30, setup new pc, optimalize fow refresh/update	3	9	121	24,2	200,0	−79,0
04.03.2020		6,5	0,5				9-16, spatial indexing for who to update fow of	7	16	128	25,6	203,8	−75,8
05.03.2020						0,5	Lecture, planning	0,5	16,5	128,5	25,7	207,7	−79,2
06.03.2020							home	0	16,5	128,5	25,7	211,5	−83,0
07.03.2020								0	16,5	128,5	25,7	215,4	−86,9
08.03.2020								0	16,5	128,5	25,7	219,2	−90,7
09.03.2020		6					9-15, optimizing DDA & FowMan, bitarray to bytearray, timer on fowupdate, list -> hashSet	6	6	134,5	26,9	223,1	−88,6
10.03.2020		6,5					9:30-16:00, optimizing, Parallell ForEach, profiling	6,5	12,5	141	28,2	226,9	−85,9
11.03.2020		6,5	0,5				9-16, optimizing, profiling, remove excess code, FowByteArr -> index List -> color array directly	7	19,5	148	29,6	230,8	−82,8
12.03.2020		3			4		9-16, minor optimals, research ESC + Unity Jobs	7	26,5	155	31	234,6	−79,6
13.03.2020	0,5						home	0,5	27	155,5	31,1	238,5	−83,0
14.03.2020								0	27	155,5	31,1	242,3	−86,8
15.03.2020								0	27	155,5	31,1	246,2	−90,7
16.03.2020							home, study	0	0	155,5	31,1	250,0	−94,5
17.03.2020							home, study	0	0	155,5	31,1	253,8	−98,3
18.03.2020							Exam economics	0	0	155,5	31,1	257,7	−102,2
19.03.2020		3			3		work home, setup, ECS	6	6	161,5	32,3	261,5	−100,0
20.03.2020					5		work home, ECS errors	5	11	166,5	33,3	265,4	−98,9
21.03.2020								0	11	166,5	33,3	269,2	−102,7
22.03.2020								0	11	166,5	33,3	273,1	−106,6
23.03.2020					6		work home, 10-16, Entity errors	6	6	172,5	34,5	276,9	−104,4
24.03.2020					6		work home, 10-16, importing entities still	6	12	178,5	35,7	280,8	−102,3
25.03.2020					6		work home, jobs/entities	6	18	184,5	36,9	284,6	−100,1
26.03.2020		4			2		work home, parallel jobs experiment	6	24	190,5	38,1	288,5	−98,0
27.03.2020		5					work home, minor optimizations	5	29	195,5	39,1	292,3	−96,8
28.03.2020								0	29	195,5	39,1	296,2	−100,7
29.03.2020								0	29	195,5	39,1	300,0	−104,5
30.03.2020	3	3		1			work home, 9-16, list->array, spatial indexing (getFOW), find edges -> make line directly, main report setup, 2 emails	7	7	202,5	40,5	303,8	−101,3
31.03.2020	1	2	1		3		work home, 9-16, ESC testing, meeting summons & report 4, meeting 4, 1 central DDA bank	7	14	209,5	41,9	307,7	−98,2
01.04.2020		7					work home, 9-16, DDA job testing	7	21	216,5	43,3	311,5	−95,0
02.04.2020		7					work home, 9-16, DDA job working, optimizing	7	28	223,5	44,7	315,4	−91,9
03.04.2020		7					work home, 10-17, DDA job optimizing (spatial indexing)	7	35	230,5	46,1	319,2	−88,7
04.04.2020							Easter start	0	35	230,5	46,1	323,1	−92,6
05.04.2020								0	35	230,5	46,1	326,9	−96,4
06.04.2020		4					Working on the side, (un) Concatinator job / Parallel.For (parallelizing de-construction of RunJobs results array	4	4	234,5	46,9	330,8	−96,3
07.04.2020		4					Parallelize setting color-array (thread safe)	4	8	238,5	47,7	334,6	−96,1
08.04.2020		2			2		Main report work / research	4	12	242,5	48,5	338,5	−96,0
09.04.2020		2			2		Main report titles according to template, research other thesises	4	16	246,5	49,3	342,3	−95,8
10.04.2020								0	16	246,5	49,3	346,2	−99,7
11.04.2020								0	16	246,5	49,3	350,0	−103,5
12.04.2020								0	16	246,5	49,3	353,8	−107,3
13.04.2020							Easter end	0	0	246,5	49,3	357,7	−111,2
14.04.2020	7,5			0,5			work home 9-15 & a little on the evening, write weekly reports 8-13, disposition, email supervisor, clean and comment code	8	8	254,5	50,9	361,5	−107,0
15.04.2020	8						work home, clean and comment code, system documentation, sketch class diagram	8	16	262,5	52,5	365,4	−102,9
16.04.2020	6				2		work home, system document, research testing in visual studio, class diagram, architecture sketch	8	24	270,5	54,1	369,2	−98,7
17.04.2020	8						work home, class diagram, architecture diagram, system doc	8	32	278,5	55,7	373,1	−94,6
18.04.2020								0	32	278,5	55,7	376,9	−98,4
19.04.2020								0	32	278,5	55,7	380,8	−102,3
20.04.2020	4			0,5	3	0,5	Work home, research generate api, sent mail system doc, planning, main report introduction, research theory	8	8	286,5	57,3	384,6	−98,1

Date	Documentation	Programming	Meetings	Emails	Research	Other	Note	Total day	Total Week	Total acc	Total percent	Planned acc	Dif
21.04.2020	8						System doc: classes description, diagrams touchup, testing, start API documentation	8	16	294,5	58,9	388,5	−94,0
22.04.2020	8						Finish system doc, finished API doc.	8	24	302,5	60,5	392,3	−89,8
23.04.2020	8						Theory: DDA+Bresenham	8	32	310,5	62,1	396,2	−85,7
24.04.2020	7,5			0,5			Theory: Bresenham + Spatial Index + Burst, send mail	8	40	318,5	63,7	400,0	−81,5
25.04.2020								0	40	318,5	63,7	403,8	−85,3
26.04.2020								0	40	318,5	63,7	407,7	−89,2
27.04.2020	8						Theory: Game engine, RTS	8	8	326,5	65,3	411,5	−85,0
28.04.2020	8						Theory: Fog of War, Line of Sight, Burst Compile	8	16	334,5	66,9	415,4	−80,9
29.04.2020	7,5			0,5			Cleanup theory / intro, send mail, figures, begun tech: Unity+C#+Dev process	8	24	342,5	68,5	419,2	−76,7
30.04.2020	8						Tech: Vision, drawing lines, iteration	8	32	350,5	70,1	423,1	−72,6
01.05.2020	8						Intro + theory done! Tech: Parallelization	8	40	358,5	71,7	426,9	−68,4
02.05.2020								0	40	358,5	71,7	430,8	−72,3
03.05.2020								0	40	358,5	71,7	434,6	−76,1
04.05.2020	8						Results: Scientific, Engineering	8	8	366,5	73,3	438,5	−72,0
05.05.2020	8						Results: Administrative, bullet points dicussion	8	16	374,5	74,9	442,3	−67,8
06.05.2020	8						Discussion: Scientific, Engineering, Admin	8	24	382,5	76,5	446,2	−63,7
07.05.2020	8						Discussion: Effort, Ethics	8	32	390,5	78,1	450,0	−59,5
08.05.2020	8						Tech + Results + Discussion done	8	40	398,5	79,7	453,8	−55,3
09.05.2020	1						DDA pseudocode	1	41	399,5	79,9	457,7	−58,2
10.05.2020								0	41	399,5	79,9	461,5	−62,0
11.05.2020	8						Clean up results, discussion, start conclusion	8	8	407,5	81,5	465,4	−57,9
12.05.2020	6				2		LaTeX Overleaf start, preface + task + abstract, weekly reports 14-17	8	16	415,5	83,1	469,2	−53,7
13.05.2020	8						Main report to Latex overleaf conversion	8	24	423,5	84,7	473,1	−49,6
14.05.2020	8						Conclusion + abstract + task + preface + references done, finishing touches	8	32	431,5	86,3	476,9	−45,4
15.05.2020	8						attach appendices, finishing touches 2, finish report	8	40	439,5	87,9	480,8	−41,3
16.05.2020								0	40	439,5	87,9	484,6	−45,1
17.05.2020							Weekly report 18	0	40	439,5	87,9	488,5	−49,0
18.05.2020	8						presentation	8	8	447,5	89,5	492,3	−44,8
19.05.2020	8						presentation finish	8	16	455,5	91,1	496,2	−40,7
20.05.2020	4						DEADLINE, weekly report 19	4	20	459,5	91,9	500,0	−40,5
SUM	233,5	144,5	8	5,5	58	10		**459,5**			min		
											500	.+-50	



NTNU

Kunnskap for en bedre verden