Universidad San Jorge

Escuela de Arquitectura y Tecnología

Grado en Ingeniería Informática

Proyecto Final

Design and implementation for discretization of user-generated patterns

Autor del proyecto: Jorge Lacort Navarro

Director del proyecto: Antonio Iglesias Soria

Villanueva de Gállego, 11 de septiembre de 2020

Este trabajo constituye parte de mi candidatura para la obtención del título de Graduado en Ingeniería Informática por la Universidad San Jorge y no ha sido entregado previamente (o simultáneamente) para la obtención de cualquier otro título.

Este documento es el resultado de mi propio trabajo, excepto donde de otra manera esté indicado y referido.

Doy mi consentimiento para que se archive este trabajo en la biblioteca universitaria de Universidad San Jorge, donde se puede facilitar su consulta.

Firma	Fecha
	11/09/2020

Agradecimientos

Gracias a mi familia por el apoyo y especialmente a mi padre y a mi madre durante todos estos años, nada de esto hubiera sido posible sin ellos y me siento afortunado de ser su hijo y de que estuviesen a mi lado incluso a través de los múltiples baches que he pasado a lo largo de mi carrera universitaria, no creo que hubiese tenido la motivación para acabar este sueño si no fuese por ellos.

Por supuesto también gracias a la universidad y a todas las personas que han hecho todo esto posible, con profesores metidos en la industria que me han ayudado a tener una mejor perspectiva sobre la situación actual y los desafíos que conlleva el desarrollo de un videojuego, y compañeros de clase con los que ha sido un placer trabajar y que no creo olvidar fácilmente.

Antes de meterme en el doble grado de ingeniería informática y diseño y desarrollo de videojuegos, en principio sólo pensaba hacer la carrera de informática, y si no fuese por mis compañeros y algunos profesores, ni se me hubiera ocurrido meterme también a hacer la carrera de videojuegos, y la verdad es que me ha ayudado a apreciar más esta pasión mía y a la gente que trata de crear estos productos multimedia tan elaborados.

Contents

Resumen

Diseño y desarrollo de mecánica de juego para definir formaciones de unidades en videojuegos RTS, permitiendo al usuario dibujar una línea como una cadena poligonal en el escenario del juego, que se convierte en una formación de unidad, rellenando la línea con todos los elementos seleccionados y expandiéndola al crear nuevas filas de unidades alrededor de la línea en caso de que sea necesario. El principal problema es diseñar una solución que funcione para cualquier línea dibujada por el usuario en tiempo real y con una complejidad en tiempo O(n^2). Al ser una novedad en el género RTS, esta mecánica de juego podría ayudar al atractivo comercial del videojuego en el que se ha integrado.

Aunque ya es difícil encontrar un diseño que funcione para todos los casos, tener en cuenta que el usuario pueda dibujar líneas tan complicadas como quieran con el límite de complejidad en tiempo requerido, es lo que hace este problema particularmente complicado. En este documento presentamos una solución implementada, aunque sería necesario mejorarla antes de su lanzamiento al mercado.

Abstract

Design and development of a game mechanic to define unit formations in RTS videogames, allowing the user to draw a line as a polygonal chain in the game scenario, which becomes the unit formation, filling the line with all the elements selected and expanding it by creating new rows of units around the line if needed. The main issue is to design a solution which works for any possible line drawn by the user in real-time and with a time complexity limit O(n^2). Due to its novelty in the RTS genre, this game mechanic could be a selling point for the videogame in which it has been integrated.

While finding a design which works for all cases is difficult by itself, having both the possibility of the user drawing lines as complex as they want and the limit in time complexity required, is what makes this problem particularly challenging. In this document we present a solution implemented, thought, it would still need some polishing before being released.

1.1. Keywords

RTS, real-time strategy, unit formation, polygonal chain, pattern, visual pattern, geometry, computational geometry, graph theory, cycle, chordless cycle

2. Introduction

This project is going to be used in a real-time strategy videogame, which we will call onwards by using its abbreviation, RTS. As there are multiple definitions of the strategy videogame genre, we are going to define them as Katie Salen and Eric Zimmerman mentioned (1), saying that the main differentiation of strategy games compared to others is the potential for a lot of choices and them being the focus of the gameplay, therefore RTS could be defined as strategy videogames which are played by all users simultaneously, and being more precise in the case of the videogame for this project, RTS is a "…time-based game that centres around using resources to build units and defeat an opponent…" (2).

While there was a time where strategy games were one of the most popular videogame genres, especially when it started, its popularity has been fading since then, and while it still makes sales, nowadays being the 8th best seller genre with around 3.7% (3), it is far from its origins. This can be related to many reasons, like the rising in popularity of other videogames genres like shooters, which have become more viable as technology has improved, or the low competitiveness that there has been, with fewer companies investing in strategy videogames and the most popular nowadays being remakes of videogames made two decades ago, but another factor which could have influenced the current situation is that there has not been many changes or improvements made to the original formula, something reflected when studying the state of the art.

What have been researched for this project is the definition of unit formations, something which has not changed much since the beginning of the RTS genre and is essential to it. The study found some cases in which there has been videogames doing formations slightly different than usual, however only finding one videogame doing something like what is being suggested in this document.

The idea is to mix the freedom and creativity of drawing lines with the generation of formations, therefore the user can draw a shape on screen with one trace, and the system generated must be capable of recognizing the line drawn and insert a group of units in the line, creating new rows of units around the line if needed. However there is a constraint to all this, since this is a real-time strategy game, the system must be capable of solving the problem in real-time and if possible, there should not be a noticeable delay between the user drawing the shape and the units moving to the new position. For that, the time complexity of the algorithms used is capped to $O(n^2)$ and if possible it must be capable of solving it with hundreds of units at once.

Aside from the videogame found, which due to its simplicity with what is being asked for this project does not compare, there has not been found any specific solution to this problem, therefore most of the effort in this case is figuring out a design which can work as best as possible. From all the designs debated, this document shows some of those designs and talks in more detail about the one which was finally implemented.

The following pages explain the process of research, design, implementation, and final integration of the project to the videogame. The state of the art explores how RTS unit formations and drawing mechanics are used in other videogames, objectives show the initial and final scope with all the features planned to deliver, methodology explains the development method used and the results of using it, economic study calculates the cost of the project, implementation talks about some of the designs which were planned and describes the final developed solution, results shows how that final solution performs once integrated to the videogame, and in the conclusions we present our thoughts on the results and the whole project.

3. State of the art

The development of smartphones and multi-touch screens has led to an increased interest in the interpretation of patterns drawn by a user in screen, since that is the only interface mobile users can use. From detecting multiple fingers to grab something or zoom in/out, to moving a finger through the screen to scroll up/down a page, finding those patterns has become essential in the process of improving user experience on those devices, as in many cases it is easier to use and understand than adding buttons when done correctly.

Same applies for videogames, where the most popular ones tend to have mechanics tied up to recognition of movements and patterns done by the user in the screen instead of showing buttons to press on the screen. For some examples there is *Angry* Birds which requires touching the bird to be launched from a slingshot and dragging it to choose the angle and strength or the shot (Figure 1), or something simpler like dragging one element from one position to another in *Candy Crush* to move it.

Figure 1 Example of Angry Birds, the left picture shows represented as a transparent white circle how at the start the player taps and holds in the bird on the slingshot, while on the right he has dragged his finger to aim, launching it after releasing the touch on the screen

However finding games focused on doing drawings becomes more complicated, and specially for a RTS game, that makes it hard to find any kind of meaningful research similar to the mechanic being developed in this project, for that reason this state of the art focuses mainly on researching about different solutions to the problem presented here, which in this case is making formations for RTS games and other games utilizing drawing as a mechanic.

Therefore the following pages are going to explain the different methodologies found being used in these games and at the end will try to come up with some conclusions to understand what users might expect from an RTS videogame with drawing mechanics.

3.1. Making formations in RTS videogames

As explained in the introduction, in RTS videogames are about building units to defeat an opponent, and to make that possible, usually the user also must control those units. The most basic method to control them is by moving them from one position to another, however it is necessary to consider that there is usually more than one unit to move, therefore there are multiple agents sharing the same scenario to calculate pathfinding, which leads to the problem of having to figure out collisions between units as they all try to go to the same point, and that can also affect the gameplay as the behaviour of the units change.

For that reason, designing a solution to this problem becomes an important element in RTS videogames, as it affects how users are going to manage their units and whether the gameplay is more focused in building and resource management or in managing units. This part therefore tries to identify the different kinds of group formation designs by researching the videogames released to the date. These are the identified categories:

Figure 2 Example of Clash Royale, showing how the units automatically go to the opposite side and attack by themselves

- **No control of units:** The user does not control the movement of the units; in this category there are for example tower defense videogames like *Clash Royale* Figure 2 (4), where the units cannot move or automatically move to attack or to some degree tycoon videogames like *Simcity* or *Two Point Hospital*, where NPCs do their actions by themselves automatically, and the user has to focus on the management of the business. They completely remove unit interaction as the only thing that matters is resource management and the tactics made over time.

Figure 4 Example of Command&Conquier: Tiberian Sun, units can be seen grouped randomly towards a point in the red circle

- **No formations:** When talking about the classical warfare videogames, this was the initial approach done by many of them, instead of grouping the units in a formation and giving them a position to go and occupy, this method consists on doing the pathfinding to the point indicated and having a stopping condition for when a unit blocks the path towards the point with units grouping around that point (Figure 4), at the beginning this was a messy technique which caused some pathfinding problems, but over the years solutions were found to make it work better as explained by *Louis Castle* of *WestWood Studios* (5). There are many videogames in this category, the *Command&Conquer* saga being the one which popularized it, other examples are *Age of Empires 1*, the *Warcraft* and *Starcraft* saga, and more recently videogames like *8-Bit Armies* or *They Are Billions*. It is the most popular and simple system, most of these games tends to add complexity to unit management by designing different types of units with different strengths and weakness.

Figure 3 Example of Cossacks 3, showing the rectangular formations of multiple groups of units

- **One formation:** Units are forced to move towards the point in a formation as a group, sometimes the units are always grouped from the beginning and in other cases they move until they make the formation and move towards the point, usually the shape of the formation tends to be a rectangle (Figure 3). What helps in this case is that only one unit needs to go to the point, and the others stop at a point relative to that unit which still maintains the formation. There are city building games like *Pharaoh*, *Zeus*

and *Stronghold*, which are mostly Tycoon games but still have some war features, and also more classical RTS like *Rise of Nations*, *Warhammer Dawn of War*, the *Cossacks* saga, *Age of Mythology*, the *Halo Wars* saga or more recently *BANNERMEN*. This adds a bit more of complexity to the management of units, as the engagement results can vary depending on how the groups enter battle, but it is still not as important as other elements.

Figure 5 Example of Age of Empires II: HD Edition, in this case there is a mangonel on the left and archers on the right, the archers start in a rectangle formation, but as the mangonel shoots, they change to a split formation evading the attack

- **Multiple formations:** The same as with one formation applies, but in this case the group can use differently shaped formations, which the user can choose at any time. In this case there are videogames like *Age of Empires 2 and 3*, *Praetorians* and *Empire Earth*. This adds a layer of complexity to the management of units, especially when the shape of the formation leads to different results in the battles as shown in Figure 5 (6).

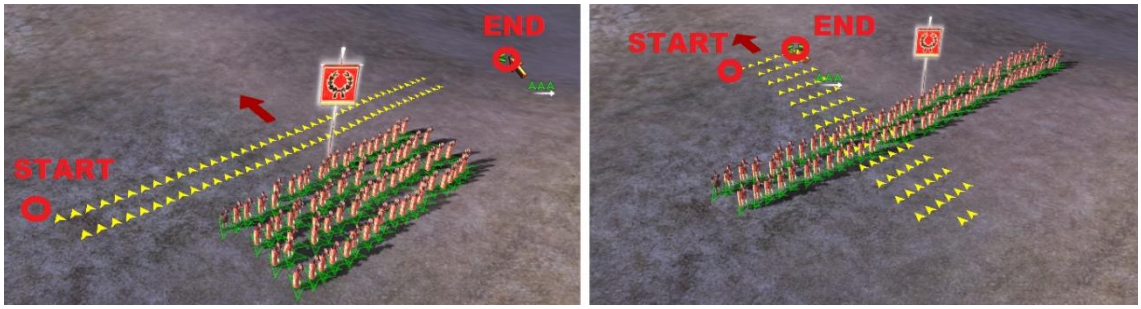


Figure 6 Example of Rome Total War, showing the start and ending point of the cursor while the player was dragging it and showing how the formation shape changes and the direction is perpendicular to that

- **Variable sized formations:** This is similar to one formation and usually units follow the same group all the time, but in this case when moving the units, the user clicks and drags from one position to another, drawing a line which indicates where the first row of the group stands, with the rest of units following behind, the further the position of the ending point, the more units each row occupies and vice versa, usually the formations follow a rectangle shape. This is a signature mechanic of the *Total War*

saga, with matches only focused on unit management. In this case the user not only chooses the shape of the formation, but also the direction the units are facing (Figure 6), and in the *Total War* saga this is especially important as the direction of the units affects their performance. While all the examples before this follow a point and click approach, in this case the user has to do more steps to perform a movement, having more drastic effects in the outcome of the fights, making this mechanic of the pillar of the gameplay for these videogames.

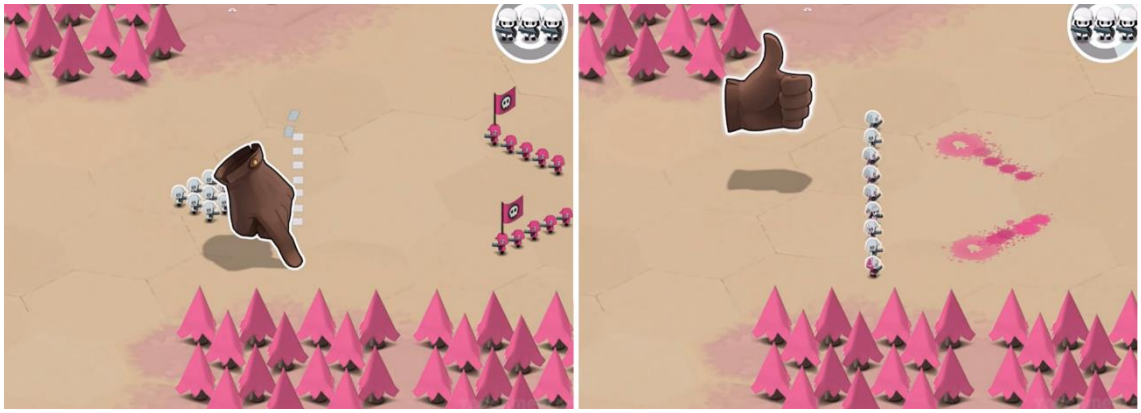


Figure 7 Example of Tactile Wars, in this case the player draws a line and the units fill that shape drawn by the user, though it could be a different and more complex shape

- **Drawn formation:** This is the system used for this project, the user draws a line in the screen, which is represented as an array of points, and the videogame tries to guess which one is the formation which the user wanted to use (Figure 7), due to its freedom of use, this is the system which is more open to interpretation and therefore requires more thought put into the design, but also for that same reason it is difficult to find an example of a videogame using it. Only one game was found using that system called *Tactile Wars*, a mobile tactical game in which the user only focuses on moving a group of units from one point to another of a scenario while fighting opposing forces. In this system as with the last one, the mechanic is treated as the core of the videogame.

Figure 9 Example of Tooth and Tail, in the first image the user calls his troop to group with the commander (the character with a flag), and as seen in the second image they group around the commander

- **Following a character:** In this game the movement of the units revolves around following the commander character used by the user, there is different solutions to this but overall they take the commander character as the reference point to guide their position in the group (Figure 9). In this case there is mixed RTS/Action games like *Shieldwall*, the *Pikmin* saga and the *Overlord* saga, where the user can guide his units but he can also use his character in a 3rd person point of view, though there also exists pure RTS videogames like Tooth and Tail where the commander character does not act directly. In regard to complexity of mechanics, this is the most simple and intuitive to utilize by users, especially for consoles using a controller where a the point and click system is not as effective, at the same time it becomes more complicated to give complexity to the mechanic as everything is tied up to the main character.

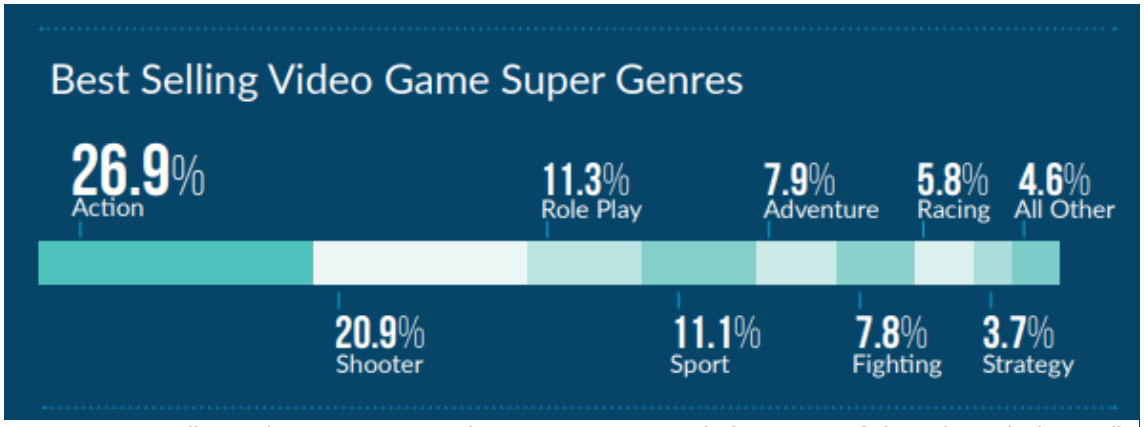


Figure 8 Best-selling videogames in 2018 by genre, strategy only has 3.7% of the sales, which is still a good amount, but there are other genres above it

With the exception of drawing formations, all these different designs of the mechanic to group units in strategy games had been already invented decades ago, as it has been something which has not evolved in all these years, maybe adding some variations or polishing the results. This probably is related to a couple of factors, first of all the designs

of these mechanic have proved to work all these years, so there is no reason to invent new methodologies, on the other hand the RTS genre, while one of the most popular when it started, it has been losing popularity over time as shown in Figure 8 (3), specially the classic RTS games, which nowadays mostly see releases of remasters or remakes of the older games, so there has not been reason to try and change the system so far.

3.2. Drawing game mechanics

Finding games using drawing mechanics is not that easy, the main reason probably is because it is hard to implement it in a satisfying and meaningful way, as it requires a system based on prediction and/or recognition of visual patterns to work, there are many games in mobile which have to recognize what the user is drawing in screen, but usually they are very simplified systems which just need to recognize a condition or two from the input of the user.

While the examples are scarce, they are not only found in mobile games and it is possible to find them in PC or consoles, the following points show some of the cases found by categories depending on what is done with the input of the user:

Figure 10 Example of Pokemon Ranger, the blue line represents the circle the player is drawing and in yellow is the past circle drawn, in pink is the pokemon being captured

- **Meet a condition:** This is the most basic system and the most widely used by mobile games, there can be many version of this and specially tailored for each videogame so instead will give a couple of examples where what is shown as output is dependant to what the user has drawn. First there are videogames like *Fruit Ninja*, which only needs to record when the user first touched the screen and when he released it, and using those two points drawing the line which connects them to draw a slice on screen to cut the fruit. On the other hand, there is something like *Pokemon Ranger*, which requires drawing circles around the Pokemon to capture, and the faster the user does it, the more effective it is. To make it work just detect the position the user is touching and calculate the angle relative to the Pokemon in polar coordinates, if the value keeps increasing and does 360º a circle has been done (Figure 10). These are the easiest drawing mechanic to implement but also the most effective and popular.

Figure 11 Example of Okami, drawing a circle to interact with the element contained in such circle, the system is recognizing that the user drew a circle instead of a line or something else for that

- **Pattern recognition:** Probably one of the most complex but also well documented due to its common use for other purposes aside from videogames in the field known as computer vision. In this case the user draws something, and the system must be capable of recognizing what the user drew by comparing it to shapes stored in the system, trying to find if the pattern is close enough to one of the shapes. Also, it is probably one of the oldest drawing mechanics, as it was used by the videogame *Black&White* released in 2001, another videogame using that and the *Fruit Ninja* system of cutting things is *Okami* (Figure 11). While this gives a lot of expressivity to the user when playing, it is also probably one of the most obtuse systems as the user has to guess which drawing are recognize as a shape or another, and also it is very dependent on how efficiently the pattern recognition system is implemented to work properly.

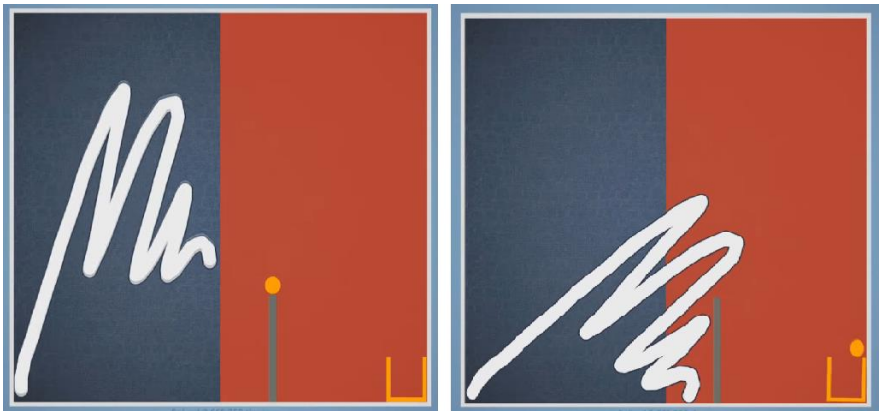


Figure 12 Example of Bring it On, on the first image the player has drawn the white shape, and using in-game physics the shapes moves accordingly hitting the orange ball to get inside the cube on the right

- **Filling shapes drawn:** The user draws a shape and an object with that shape appears in the videogame, usually a line with some width is drawn. Some examples of this are *Kirby: Canvas Curse*, a platform where the user must draw the path to follow*, Car Drawing Game* where the user draws the shape of the car to use, and *Brain It On!* A game about physic puzzles (Figure 12). This mechanics work particularly well with physics-based systems, where the drawing gives a lot of interactivity as it is affected by the game physics.

Figure 13 Example of a Unity project in development, showing first the shape drawn by the player and later how the units try to fill that shape

- **Filling a line with finite elements:** Which is the one implemented in this project, given an array of points in order representing a line as a polygonal chain, fill it with the finite number of elements given, extending the line in whatever way desired if necessary (Figure 13 _(7)). Only example for this is the same videogame as the one explained in RTS formations, Tactile Wars, it was not possible to find any other example using this. While this videogame shares a similar behaviour to this project, there are also some differences due to the videogame requirements, being simpler than what is being solved in this document. Given the rarity of this system, it is probably a good idea to use it as a differentiation with other videogames of the genre, though the main problem here probably is usability of the system, something which is clear in Tactile Wars as they limit the amount of things the user can draw a lot.

The simplest the idea for drawing mechanics, the more widely it is used in videogames, finding them more commonly in mobile gaming nowadays, however, using more complex systems does not seem to be a bad idea, as most of the videogames found using those were fairly successful, of course this could also be related to other aspects of the videogames studied here, but the drawing mechanics seems to be the main attractive factor they have. Checking the numbers of the mobile videogames discussed in Google Store, *Brain It On!*, being the most successful, has

>10 000 000 downloads and a score of 4.4 (8), *Tactile Wars* has over >5 000 000 downloads and a score of 3.9 (9), and even a simplest and less polished videogame such as *Car Drawing Game* has >50 000 downloads and a score of 3.7 (10).

3.3. Conclusions

After doing research of the current RTS videogames in the market, using a drawing mechanic like the one implemented in the videogame of this project seems to be a rarity, as the only videogame found was Tactile Wars, which was fairly successful back in the days it released as shown by the data of the store, this can be used as a proof of concept, showing that it is possible to make it work and marketable, probably one of the main things to take from it is that simplicity, and the synergy with other mechanics of the videogame, was probably the key to its success.

From the other videogames researched, both using and not using formations seems to be good ideas by themselves, as there are many found cases of successful RTS videogames implementing them, also from them we can guess that users may expect certain group behaviours in their RTS videogames, with the most common either being the mob grouping as shown in the ones not using formations, where units just group around the highlighted points, or rectangle formations which were the most common found in the ones using formations, that is something to take into consideration when designing the behaviour of units when making formations following a line.

From the research done to drawing videogames, the main point to consider is that the best approach to designing this mechanics is to keep it simple. The most commonly used and popular implementations tend to be the ones which do not try to do something complicated to understand, though that does not mean that their implementation is simple, but is more related about the complexity of the mechanic from the point of view of the user, therefore it is better to make systems with easily predictable outputs.

4. Objectives

The main objective of this project is that, given a variable number of traces drawn by a user in a bidimensional map; distribute a variable number of elements (soldiers) accordingly in a manner which is intuitive so that the user can create his own military formations.

The following points will talk first about why the project was changed, the original objectives from the proposal, the different features which were debated to be implemented, the initial defined scope and finally showing any changes to the objectives done during the development of the project.

4.1. Change of project

Before talking about this project, it is necessary to mention the change of project which was made. The original project, called "Creación de un sistema de inteligencia artificial para agentes independientes en entornos cambiantes", was being done in collaboration with the publisher *Sindiecate* for the videogame developer company *Brainwash Gang*. The project objective was about enhancing the AI system of a videogame.

However the company went through some issues and the project ended up being discarded for a while, and when they went back to it, they did not need the feature anymore as they had completely changed the videogame. As it was necessary to request them tools for the development of the feature, it became impossible to continue the project without starting again, losing all the work which have been done. Therefore, the choice was to start with another project, as that would be less problematic than trying to recover the original one.

4.2. Objective

The objectives defined for the project are:

- Design an algorithm which receives a sequence of point from drawn traces by the user, a quantity of elements to distribute along the pattern and the space occupied by each elements, and it should be capable of distributing the elements evenly to the sequence of points given. The algorithm will expand or shrink the geometric shape depending on the number of elements given and their separation. Also, the elements will be distributed between the different trails in an adequate and proportional way.
- Make the implementation of the algorithm in Unity so that it can answer in real-time to all the possible cases to generate. Study the limits and test the efficiency of the algorithm, making changes if needed to solve the problem for the quantity of units required for the videogame.

- Integration of the algorithm in a real case of a system which generates unit formations in a videogame already in development.

4.3. Defined features

As the features were not defined in the proposal of the project, a reunion was made with the company *Kraken Empire* to discuss the different features which would be required for the videogame, below shows a list of those features in order of importance:

- **Distribute elements in a line:** Given a variable number of elements, the space occupied by each of the elements and a variable number of bidimensional spatial points representing a line, distribute the elements following the line shape, generating new rows around the initial stroke if there is not enough space in the original line.
- **Recognize close/open lines:** Recognize whether the line drawn has been closed at the end or not, therefore giving a different response in each case. While the elements should try to fill the line when it is open, if the line is closed the elements should try to fill the shape drawn.
- **Distribute in multiple lines:** Given more than one line, it should be capable of distributing the elements through each of the lines accordingly, with each line using the same features as described before.
- **Detect patterns:** Recognize different basic geometric patterns and give a different response for each case. If the shape drawn represents closely a basic geometric shape like a circle, a rectangle, or a triangle, that is the shape which should be filled.

The idea at the end is to have a function which can be integrated in the videogame, so that if the user selects a group of soldiers and draws shapes, the function answers with a distribution of the soldiers in formation, it must be functional, consistent and easy to understand for the user.

4.4. Initial scope

To find the scope of this project, each feature was divided in all the steps required to make it possible and an estimation of the time required to develop them was made. After doing that, the feature of detecting patterns was calculated to require too much time and was left out of the scope of the project, as it can be seen in the scope shown in the following page:

Project Phase	Scope	Hours
Basic implementation	First draft of memory (State of the art, objectives, methodology)	40
	Integration of tools to project	8
	Design and base code	8
	Subtotal	**56**
Distribute elements in a line	Study possible distribution of elements	8
	Implement best ideas	40
	Second draft of memory (Implementation, results)	8
	Subtotal	**56**
Recognize close/open line	Initial definition of limits	4
	Implementing differentiation	24
	Test and fix limits	4
	Third draft of memory (Implementation, results)	8
	Subtotal	**40**
Solve extreme cases	Find extreme cases	8
	Solve extreme cases	24
	Fourth draft of memory (Implementation, results)	8
	Subtotal	**40**
Distribute in multiple lines	Design elements per line	8
	Implement distribution	24
	Fifth draft of memory (Implementation, results)	8
	Subtotal	**40**
Finish project	Test and fix issues	20
	Finishing memory (Summary, introduction, economic study, results, conclusions)	48
	Subtotal	**68**
	Total	**300**
OUT OF SCOPE		
Detect patterns	Study pattern recognition	16
	Implement patterns	40
	Test and fix limits	16
	Draft of memory (Implementation, results)	8
	Subtotal	**80**

4.5. Final scope

After going through the development of the project, many changes were necessary to be done, as the problem was more complex than planned at the beginning, increasing the time required to distribute the elements in a line and solve extreme cases, while giving priority to extreme cases over recognizing close/open lines. And while the feature of recognizing close/open lines was studied, it was not implemented as it was not viable after the investigation. In the case of distributing in multiple line it came more to the realization that it was possible to implement a simple solution to solve that problem without investing that much time, so that part, as other finishing details, was implemented in the lastly added phase of fixing problems.

Project Phase	Scope	Hours
Basic implementation	First draft of memory (State of the art, objectives, methodology)	42
	Integration of tools to project	2
	Design and base code	8
	Subtotal	**52**
Distribute elements in a line	Study possible distribution of elements	12
	Implement best ideas	48
	Second draft of memory (Implementation, results)	16
	Subtotal	**76**
Solve extreme cases	Find extreme cases	8
	Continue development of distributing elements in a line	8
	Solve extreme cases	28
	Test extreme cases	4
	Subtotal	**46**
Recognize close/open line	Initial definition of limits	40
	Subtotal	**40**
Fix problems	Fix problems of exceptional cases	24
	Fourth draft of memory (Implementation, results)	16
	Subtotal	**40**
Finish project	Finishing memory (Summary, introduction, economic study, results, conclusions)	46
	Subtotal	**46**
	Total	**300**

5. Methodology

Due to the creative process of this project and therefore having some uncertainty in how things are going to develop, it makes more sense to use an iterative process and therefore agile methodologies, to be precise this project uses Scrum, with some adjustments to fit better the requirements.

This part is separated in three parts, briefly explaining first the Scrum methodology, then the methodology used for this project and finally showing the results of the planification.

5.1. Scrum methodology

As an agile methodology, the main point of Scrum is to allow teams to work together and have a flexible work environment which is more easily capable of reacting to changes of the objectives as the project goes on, something useful for a project which requires experimentation as this one.

To make that possible, Scrum offers a framework with a series of tools and guidelines which can be used to facilitate the organization and follow the desired process, something explained in the following points.

5.1.1. Roles

It is necessary to define the different roles required in a team to use Scrum:

- **Development team:** People developing the product.
- **Product owner:** Focused on making sure the product is delivered.
- **Scrum master:** Focused on helping the development team work properly.

5.1.2. Artifacts

Artifacts define all the things which are made throughout the development of a project while using this methodology:

- **Product backlog:** List showing the work to be done in the project.
- **Sprint backlog:** It is the list showing all the tasks to be done in a specific sprint.
- **Increment:** It is what the product is called once a sprint is done.
- **Extensions:** Tools used to analyse the project.

5.1.3. Workflow

It is necessary to define a series of events which will define the workflow of the team to implement and utilize the methodology daily:

- **Backlog grooming:** Reviewing the backlog to make changes if needed.
- **Sprint:** A repeatable fixed time-box during which a product of the highest possible value is created.
- **Sprint planning:** As one increment is finished, it is necessary to plan the next sprint.
- **Daily scrum:** A daily meeting to make sure that all the team is aligned and working on the same page.
- **Sprint review:** A meeting done after finishing the sprint to consider changes for the product.
- **Sprint retrospective:** Another meeting after finishing the sprint to consider changes in the development process.

5.2. Methodology used

When talking about development methodologies, they should be considered as guidelines and not rules, therefore, while it is recommended to follow the process they define, it is also possible to make modifications to better fit each project and development team. In this case it is not necessary to organize a team as the project is being developed by just one person.

5.2.1. Roles

In this case it is not necessary to define roles, as only one person is developing the project, the role of scrum master is not needed as it is not necessary to have someone making sure that the team is applying the methodology correctly. Therefore, the developer just does the tasks of the developer team and the product owner defined above.

Also, the client in this case is the company Kraken Studios as the project is being developed to add the feature to their videogame.

5.2.2. Artifacts

For artifacts there is no need to make changes, as defined in the Scrum methodology, at the beginning of the project the product backlog was defined with all the items required to develop all the features desired. After that and for each sprint, a sprint backlog is made selecting items from the backlog and defining them with more detail, which is updated as the project progresses.

Of course, at the end of each sprint there is an increment with a new feature implemented, and a meeting is done with the client to show the progress and to make changes in the product backlog if needed.

Finally, there is an extension used to analyse the progress of the project in this case, a Gantt chart of the sprints showing the daily tasks defined and how those delivery times have changed over time.

To do all this, an external web application was used called Trello, creating a project open to public (11), the reason to use that one is because it has all the tools required and mainly because it was the only one found being free.

5.2.3. Workflow

There are some changes needed to be made in the workflow as it is not necessary to have meetings with a team, and for the same reason there is no need to do daily scrum.

Instead of doing meetings, all the task defined in backlog grooming, sprint planning and sprint retrospective are done by the developer himself, and something similar to a sprint review is done verbally with the client of the project, showing the new increment and making changes if necessary.

For the sprints, instead of defining a constant amount of time for each one, given that this project has features requiring between 1-2 weeks of development, the time for each sprint is defined by the required to finish each of the features as defined in the scope in the project phase category.

5.3. Results

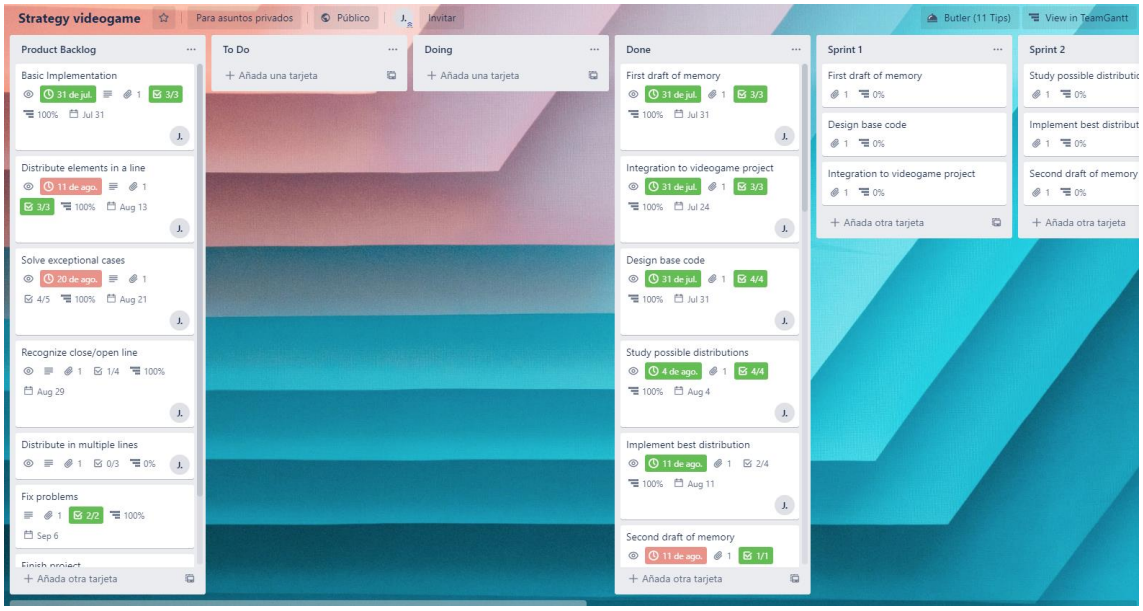


Figure 14 Trello project at the end of the development process, with the Product Backlog and the Kanban board

While lacking some features that other paid apps have, especially for extensions, Trello is simple enough which does not require much time to set things up, and still it is possible to get extensions via plugins in the same website, though the number of extensions is limited to one without paying, therefore a Gantt chart plugin was added for a better control of the project. The plugin itself connects Trello with another website called TeamGantt, with every change made in the cards of Trello being automatically updated in the chart.

The process to maintain up to date the backlog and the sprint did not require much time and it was useful to be more aware of the things which needed to be done and the time left to do them. Also, the sprint reviews (Annex A) were particularly useful to notice how the features were taking more time than planned to be finished and react accordingly by reducing the scope of the project.

5.3.1. Changes

The project started getting into a critical point of development halfway the process, as the final design was being implemented and it was becoming more complex to solve it. This is reflected in the sprints as many of the tasks defined for that period of time got constantly delayed, which made it easier to identify to act accordingly, remove the original feature of representing close/open lines, and changing it for another one to solve the problems of the initial feature.

However that did not go so well, as after studying that solution it was not good enough to be used, therefore at the end of the project, the sprint of distributing into multiple lines was

removed and a new sprint was added to make sure that at least it was possible to deliver a minimum valuable product, and that actually went quite well, as at the end it was possible to deliver a functional solution of the main feature requested.

5.3.2. Extensions

Figure 15 Initial Gantt chart of the project

Finally there are the Gantt charts mentioned before, first a Gantt chart was done in Excel to see the flow in time of how the project would go as seen in Figure 15, with all the features planned to implement at the beginning.

But of course, as there were changes done, the final Gantt chart of the project looks different to this one, this time done with TeamGantt:

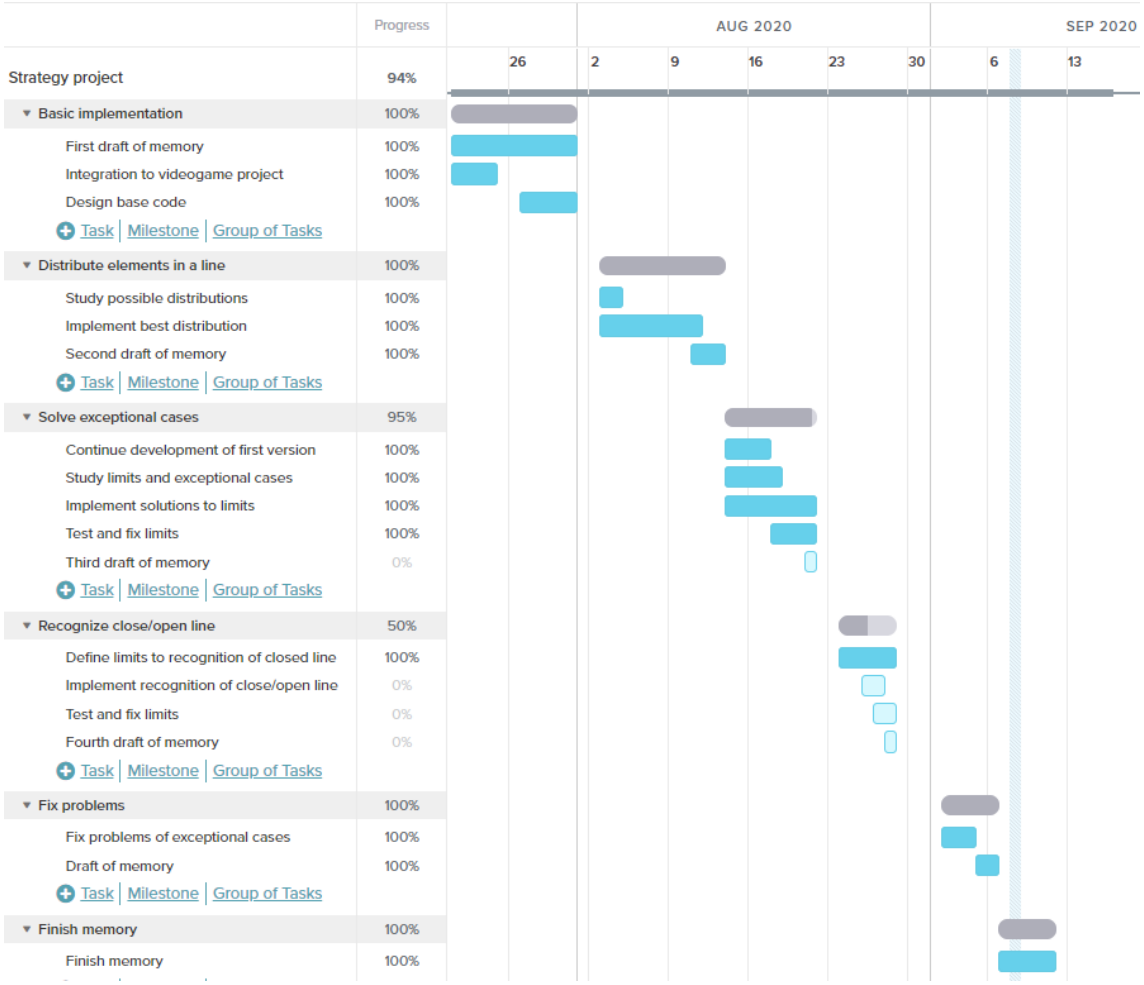


Figure 16 Final Gantt chart of the project

The bars in blue show the features which were finished while the one in a light blue are the ones which were defined but not done.

From this graph it is clear that there was a rush at the end to finish things when compared to the original chart, which was more balanced than this one, also partly because of the unstable situation of the project, there was not work done in the memory in the middle of the project, leaving all that information about the implementation to be done in the *fix problems* sprint. Of course, all the changes mentioned before are also reflected here.

6. Economic study

As there is not a way to realize a viability study of the project, since it is a feature for a videogame and there are no metrics which could be used to estimate the possible benefits, this is a recompilation of all the costs required for the design, development and integration of the feature in the videogame. The costs are divided between human resource for the people who worked on it and material resources for all the hardware, software and goods and services used.

6.1. Human resources

The people involved in the project are the following ones:

- Tutor and client from Kraken Empire, he helped providing resources and information, but also requesting the features of the project, for a total of 18 hours recorded in reunions in Annex B, and also adding another 6 hours due to many not formal reunions being done through the development process for a total of 24 hours. For the salary estimation, considering him as a client and project chief, the salary is counted as an analyst programmer.
- Author of the project, handling all the design, development, and integration of the project in the videogame, for a total of 300 hours as recorded in Annex C. For the salary estimation, considering him as a programmer with no experience working professionally in the field, the salary is counted as a junior programmer.

A worldwide employment-related search engine called Indeed has been used to estimate the costs of human resources. Doing a query for a specific job the search engine compiles all the company entries stored in their system, giving individual or the average salary. It also allows to filter by country, so as the project was offered by Kraken Empire, a company in Zaragoza, the results are from Spain. Also, as what is being used is a salary, the gross salary is estimated as 143% of the salary (12).

As the salaries on the search engine are monthly, applying the standard average monthly hours of work, 173.33 hours, and using the average salaries found in the site, the results of the human resources are:

Job	Price/hour	Hours	Cost
Analyst programmer	12€ (13)	24	288€
Junior programmer	10.5€ (14)	300	3150€
		Total:	3438€

6.2. Material resources

These are the material resources used:

Hardware			
Resource	**Price**	**Units**	**Total**
Laptop Asus GL552V	999€ (15)	1	999€
		Total:	**999€**
Software			
Resource	**Price**	**Units**	**Total**
Windows 10	145€(0€*) (16)	1	0€
Microsoft 365	69€ (17)	1	69€
Visual Studio Community**	0€	1	0€
Unity	399€/year (18)	1	399€
TortoiseSVN	0€	1	0€
		Total:	**468€**
Goods&Services			
Resource	**Price**	**Months**	**Total**
Office rent	210€ (19)	3	675€
Water	20€ (20)	3	60€
Electricity	70€ (21)	3	210€
Phone + Internet	28.95€ (22)	3	86.85€
		Total:	**1031.85€**

* Comes inside the Price of the laptop
** The free version of Visual Studio (Community) can be used by business which have less than 250 PCs like Kraken Empire

Combining all the material resources costs makes for a total of 2483.85€.

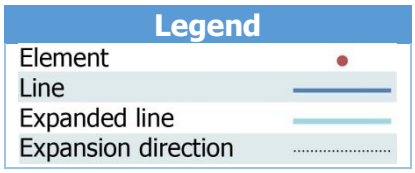
6.3. Total costs

Considering both the human and material resources required to do the project as shown above, the budget required is 5921.85€.

7. Implementation

Aside from the game *Tactile Wars*, there are no more points of reference in the subject being studied; therefore the main focus here is going to be on the design, explaining some of the main solutions thought for this problem and the reason why one design was chosen over the other.

Due to this problem being easier to understand visually, the different paradigms presented are going to be explained with the help of graphs, all of them using the language presented in the following legend:



The structure of this part is as follows:

- **Designing solutions:** All the designs which were thought for solving the main problem of distributing the elements in a line and expanding that line.
- **Solution:** Final solution implemented after choosing the algorithms for the distribution.
- **Exceptional cases:** Specific solutions done to solve some cases which the algorithms selected could not solve by themselves.
- **Recognizing open/close patterns:** Possible solution designed to solve some of the problems for exceptional cases which was not implemented.

7.1. Designing solutions

While Tactile Wars has a solution to this problem, it is done in a very limited and controlled way, with a small amount of elements (~5-30) which rarely surpasses the capacity of the line drawn and when it does, the complexity of the drawings it can do is very simple, therefore while it can work as a base idea, the design for this project requires something else. Aside from that, the complexity of the loops used in this solution cannot be more than $O(n^2)$, as it has to be capable of running in real-time, meaning the solutions must be simple and it is not possible to do many iterations to find the solution. So, the first step was all about experimentation and thinking about different solutions to the problem and the possible pros and cons.

This problem can be divided in two different parts, the first one is filling the line with elements by finding the positions to fill it, while the other is expanding the line, once the first line is filled and there is still more elements to fit in there, it is necessary to find a method to put the elements around the original line.

First it is necessary to make a definition of how the line works, because it actually is a polygonal chain, which is a connected series of line segments, though we will call it line for simplification,

but it actually is a discrete list of bidimensional points, therefore for the examples L is the line, defined by a set of points $L = [p_0, p_1 .. p_n]$ with $p = (x, y) \in Q$

7.1.1. Filling the line

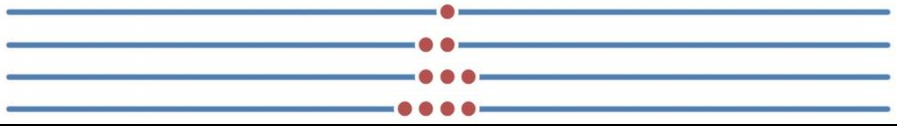
The first step is to fill the line with all the elements capable of fitting inside, which is going to be defined by the total length of the line and the space which each element occupies.

Though the main question is to think about how those elements must be distributed through the line, as the criteria for the choice of the position can vary depending on the implementation, something which also can have an effect in the gameplay of the game, as the formation of the units will be different and the results of the combat engagements can be affected by that.

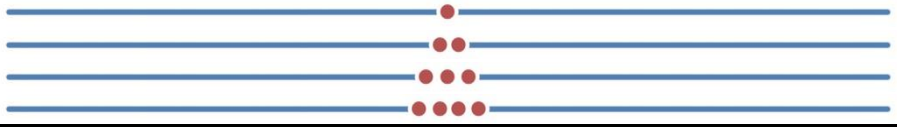
Four solutions were designed and prototyped:

In order	
Elements	**Solution**
1	
2	
3	
4	

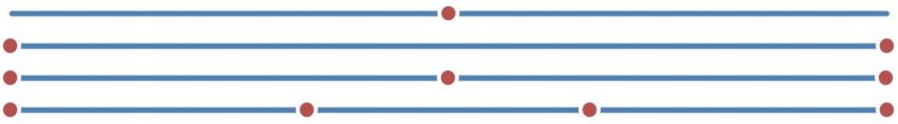
- **In order:** First design implemented, this is the simplest one, as all the algorithm must do is to travel through the line and insert an element whenever there is space for it. However, the main issue of this solution is that unless all the line is filled, this is the one which worse represents the drawing done by the user.

Centric	
Elements	**Solution**
1	
2	
3	
4	

- **Centric:** Next step, find the position in the centre and start filling it by alternatively and incrementally going left and right. This helps by doing the behaviour independent of where the user started drawing the line, however the symmetry is broken when the number of elements is even, making for an odd representation of the line on those cases.

Centric and in the centre	
Elements	**Solution**
1	
2	
3	
4	

- **Centric and in the centre:** If the main problem with centric are the even elements cases, all that is needed to improve that is adding and offset in that case to the positions. Doing that gives a better representation of the line for all the number of elements, however in all these cases, the robustness of the group formation is favoured over the representation of the drawing done by the user.

Evenly scattered	
Elements	**Solution**
1	
2	
3	
4	

- **Evenly scattered:** Final design, it works differently than the others by finding out first the positions for each element by calculating the separation between elements, and it is not much more complicated to implement than the first one. This case has the opposite effect, this is the one which best represents the drawing done by the user even for a low number of elements, however those elements can end up being more separated one from another, making more inconsistent formations, but at the same time this gives more expressiveness to the user and allows him to create more varied formations.

For this project, the representation of the line drawn by the user is favoured over anything else, therefore the best solution to implement in this case is evenly scattered.

7.1.2. Expanding the line

Once the initial line has been filled with elements, the algorithm must be capable of adding more elements to the line; however, as there is no more space in the line drawn by the user, it must be expanded by creating new position around it.

While filling the line is a very straightforward problem which can be solved in different ways, it is more complicated to figure out how the line should be expanded, as new positions which have not been defined yet must be found based on the original line, and this process must be able to work in a loop to expand even further if needed, making sure it does not get stuck in a situation where it is not capable of creating new positions.

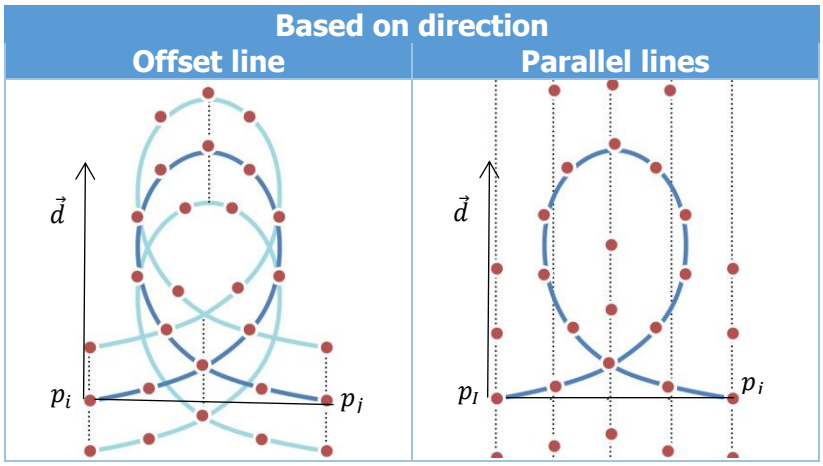
To begin finding new positions, first it is important to decide based on what those positions are going to be found, and in this case, there is two elements:

- **Elements:** Having set the first row of elements in the line, and knowing their positions, it could be possible to find positions around those elements and fill those. The main issue is finding a method for those elements to explore positions around or trying to spread them in different ways as explored in filling the line above.

- **Line:** Using the points defining the line, find new lines around the original one to fill with elements. In this case there is an added complexity of trying to solve geometry problems to define that line, however on the other hand it is possible to define shapes which more closely resemble the original line and there is the added flexibility of having a range of positions to be filled, by using the methods of filling the line.

Then, considering both approaches and after doing some designs and discarding some of them. Having the line L defined by a set of points $L = [p_0, p_1 .. p_n]$ with $p = (x, y) \in Q$, a group of elements E containing a set of points $E = [e_0, e_1 .. e_n]$ with $e = (x, y) \in Q$, and each element being a circle of radius $r_i = K$ four methods were considered for their implementation:

Based on direction



This is based on the line, using two points p_i and p_j from the original line, calculate direction $\vec{d}$ perpendicular to $\overrightarrow{p_i p_j}$, so that the line can be expanded in that direction and the opposite one. Depending on which two points are selected the results can vary, a couple of choices can be:

- **Start and end:** Choosing p_0 and p_n
- **Width based:** By defining the rectangle which encloses all the points in the line and finding the points p_i and p_j on opposite sides of the rectangle separated by the farthest distance.

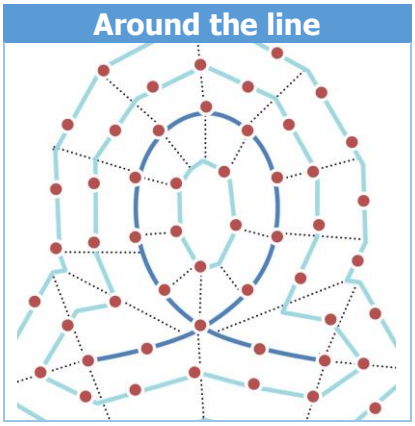
Then a method to expand the line must be defined based in $\vec{d}$, like the following examples:

- **Offset line:** Using $\vec{d}$ and $-\vec{d}$ as the direction and K as the distance, use that as an offset to copy the original line in both directions, and then fill the lines with more elements and repeat the process if needed.
- **Parallel lines:** Divide $\overrightarrow{p_i p_j}$ in equidistant points s_i separated by distance $\geq K$, then throw a line for each point s_i with direction $\vec{d}$ and each time a line collides with L,

divide the line with equidistant points p_i separated by distance $= K$. Then use the points p_i to insert more elements.

The main advantages of these methods is that they are simple to implement and they should be quick to calculate, however there are some issues with it, first of all, this method is only capable of expanding in one dimension, therefore if the line is small and it is necessary to expand the line many rows, the original drawings is going to be particularly distorted, also if the line is complex, there is a high chance that there is going to be gaps in the formation, as there can be many collisions both in offset line and parallel lines and the extended lines are not sensitive to the shape of the line in the dimension which it is not expanding.

Around the line



Also based on the line, in this case the process starts by finding the polygon $l_1 = [p_0, p_1 .. p_n, p_0]$ which surrounds the initial line l and then expanding it if needed $(l_2, l_3 \; .. l_m)$ repeating the process from the last expanded polygon. The process to find that polygon representing the line however is not that obvious and many approaches have been explored in the project for this:

- **Perpendicular:** For each pair of consecutive points p_i, p_j of l defining the vector $\overrightarrow{p_i p_j}$, find the normal $\vec{n}_i$ of the vector and from p_i, using the direction $\vec{n}_i$ and finding the point at distance K, add the point to the new line and repeat this process with all the pair of points. Repeat this with the negative value of the normal to get the line at the other side. This is a simple way of finding the polygon around by making some assumptions, which lead to the extension of the line easily being deformed by curves in the original line.
- **Line in between:** This case uses three consecutive points p_i, p_j, p_k of l, creating vectors $\overrightarrow{p_i p_j}$ and $\overrightarrow{p_j p_k}$, in this case the normal $\vec{n}_{ij}$ and $\vec{n}_{jk}$ are calculated and added $\vec{n}_{sum} = \vec{n}_{ij} + \vec{n}_{jk}$, and using $\vec{n}_{sum}$ as the direction and K as the distance from point p_j, the next point is between $\overrightarrow{p_i p_j}$ and $\overrightarrow{p_j p_k}$, and this process can be repeated in the

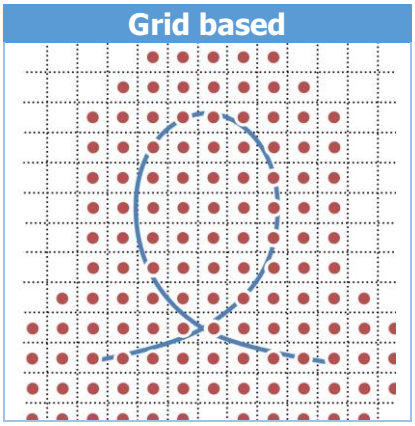
inverted order to find the line at the other side. However, for vectors $\overrightarrow{p_i p_j}$ and $\overrightarrow{p_j p_k}$ being at an angle $< \pi$, it requires to find the correct distance, as it would be $> K$ for it to be at the correct distance, and that distance can also be affected by surrounding points. This one represents more accurately the polygon surrounding the original line, however to find that distance $> K$ when the angle between $\overrightarrow{p_i p_j}$ and $\overrightarrow{p_j p_k}$ is $< \pi$ can become complicated quite easily unless assumptions are made which can lead to mistakes.

- **Circles tangent intersection:** In this case from each point p_i of l, define a circle of radius K, and for each three consecutive points p_i, p_j, p_k, find the intersection of the tangents between (p_i, p_j) and (p_j, p_k), then find the intersection between those tangents. This one has less flaws than the others, however it requires more operations to find out the solution, also still can make some odd behaviours when the angle between the tangents is $> \pi$, as the closest to a parallel those tangents go, the further the point is placed, easily converging to an infinite distance and therefore while correct, making unusable shapes in the real scenario.

These methods return lines to both sides of the initial line, however they do not enclose the line, for that it is necessary to add some extra logic to add points to the sides of l, at p_0 and p_n, adding a last point returning back to p_0, just adding a point parallel to $\overrightarrow{p_0 p_1}$ and another to $\overrightarrow{p_{n-1} p_n}$ is enough, though more points can be added to modify the shape as it expands.

They follow a closer representation of the line and they are able to expand in both dimension unlike the ones based in direction, however finding an accurate solution which always works becomes more complicated, and they can quickly start getting messy for the more complicated line shapes.

Grid based

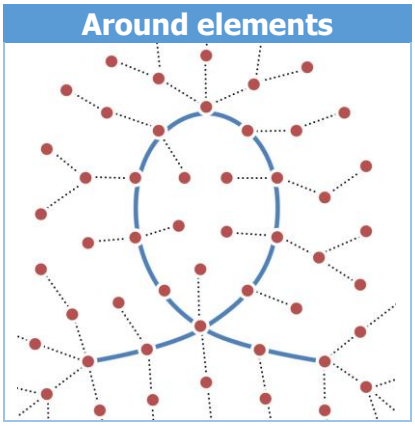


Based on the elements, the grid initial position and rotation can be calculated relative to global measures or relative to one of the points in the line, like the first point of the line, which will slightly modify the look of the formation, using global measures all the formation in the scene will have more of a grid-based looks, as if they are moving through a chess board, while doing it relative to a point of the line can make them look a bit more natural.

To make this work, it should also be used to fill the original line, the idea is to find out through which squares of the grid the line passes and add elements to that position of the grid. Once the original line is done, for each of the elements $(e_0 , e_1 .. e_n)$, check the surrounding squares in the grid and add an elements if it is empty, this process is repeated always with the last layer of elements generated until all the elements have been inserted.

While the formation using this systems looks more structured than in the others, that is also its drawback, as it can make the formation look artificial due to the grid becoming apparent, and doing some odd solutions specially for diagonals, another problem of this solution is that if one of the loops generates an extended row of elements but finishes before completing the whole row of possible positions, the formation ends up with an uneven distribution, and trying to make it not be uneven is not as trivial and may have flaws.

Around elements



Also based in elements, once the elements of the initial line l are found, for each of the elements $(e_0, e_1 .. e_n)$, either clockwise or counter-clockwise, a search of empty positions is done.

The precision of the algorithm can be adjusted by defining the number of iterations I, however that also multiplies the cost of doing the search, therefore it is recommended to start with a small number of iterations and increase it until the results are good enough.

This one gives a good distribution of elements around the line and can be effective at filling gaps, being also simple to implement, however even if the solutions is simple, the problem is the cost in time of the algorithm, as it requires first a loop for each element to find empty positions, then another loop which checks collisions with the rest of elements, and finally another loop which is repeating this process for each of the elements in the line, having a triple nested loop of cost $I \cdot n \cdot n = O(In^2)$. Therefore from all the algorithms studied here, this is the one which requires more time to be solved and it is close to surpassing the limits given for this project of $O(n^2)$ (as a constant multiplier is fine as far as it does not end up being of $O(n^3)$), though it all depends on I for that. Another problem of this solutions is something which also happens with grid based, which appears when the last row of positions for elements is not filled, and therefore spreading them becomes more complicated than with the algorithms based in the line.

Having these four methods, an order of priority was chosen for their implementation:

1. **Based in direction:** While it may be the worst at representing the drawing as it only grows in one dimension, this is the one which could have a bigger impact in the gameplay and could offer an alternative experience while also being simple to implement.

2. **Around the line:** This is the closest to the results desired for this project, the drawing representing as close as possible the drawing is the top priority, aside from that the feature of being able to spread elements around in the last line is desired, however, is it more complicated to implement than based in direction.
3. **Around elements:** Should give near close results to around the line of representing the drawing, however, it lacks the feature of spreading elements in the last line and is the most expensive algorithm.
4. **Grid based:** While it should be somewhat be able to closely represent the drawing, the blocky artificial looks is not desired for this project, preferring the formation to have a more organic shape.

7.2. Solution

After considering the pros and cons of each of the solutions and discussing it with the client, a methodology for each of the problems was selected as the favored to be developed.

For *filling the line*, the project started using the *in order* method due to its simplicity, however eventually *evenly scattered* was implemented as it represents more closely the drawing realized by the user and it is preferred over the viability of the formation created. Aside from that, having the units more separated can increase the options of the user when thinking about a strategy by allowing more varied formations.

For *expanding the line*, it went through multiple stages, at the beginning it started with a *based on direction* implementation due to its simplicity and a similar solution already being implemented in the project, however it was discarded eventually due to the problem of the algorithm not being able to adapt and grow in both spatial dimensions, not representing as accurately the drawing of the user, and increasingly deforming the formation as the number of units increases and the line length decreases, after making some tests it was decided to change for another algorithm for the videogame. *Around the line* was selected as the preferred one, the first implementation used the *perpendicular* method, however it proved to underperform with curves, increasingly deforming the drawing as new outer polygons are drawn around the original line. The next solution was using *line in between*, which represents better the drawing realized, but it handles particularly poorly extreme cases, and therefore, the final implementation uses *tangents,* using *perpendicular* for particular cases where tangents cannot be applied, like the initial and the ending point of the line or in straight lines because it is cheaper, this still has problems in extreme cases but they are a bit more manageable than the *line in between* method.

The following points show in more detail how the project works, explaining the structure of the project, the functionality of the main function and the solution for the distribution of the elements in the line using the two final methodologies selected.

7.2.1. Project structure

The result of this project has to be a function which after giving some parameters, must return a list of positions to put elements, therefore the problem could be put together in one class and solve everything in there, however, as the videogame could go through many changes, it is desirable to create a system which can be easy to change and adapt to the new requirements, and for that reason the code is divided in different classes to be more scalable.

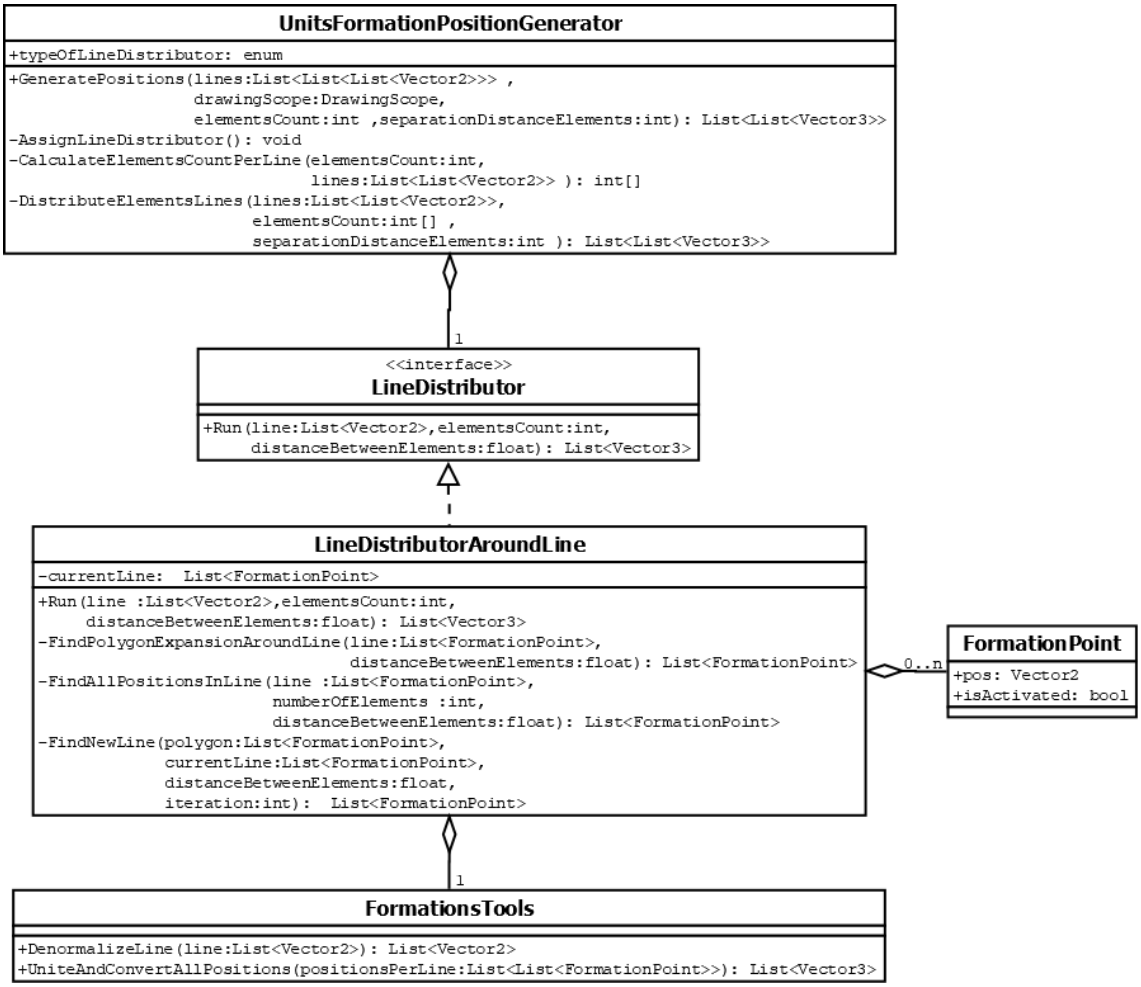


Figure 17 Diagram class of the project

The structure of the project, as shown in the diagram class above, is the following one:

- **UnitsFormationPositionGenerator:** Main class of the project and the one which interacts directly with the code of the videogame, once it receives the parameters to distribute elements in lines, it calculates the elements for each line and after selecting the algorithm for distribution, it executes them for each line and returns the distribution of the elements. Currently it only executes a *LineDistributor*, however originally it was supposed to also execute a *ShapeDistributor*, which would have controlled distributing elements for closed lines.
- **LineDistributor:** Interface implemented by all the algorithms to solve the problem of distributing the elements through a line, helping the main class to use the same logic independently of the algorithm it is using, making it easier to change things if needed. Currently it only has a *Run()* method, which gives the distribution of the elements in a line, however it can be extended if needed.
- **LineDistributorAroundLine:** This is one of the algorithms to distribute elements through a line, initially the idea was to implement multiple algorithms to test them and see which one is better, however due to the complexity of this algorithm, the scope of the project had to be reduced to just this one. While initially the project was started with a *based in direction* solution, it was quickly discarded and was still missing key features to make it work properly.
- **FormationPoint:** Originally it was necessary to save which points of the line could be used to generate an extension of the line or not, instead of using the raw list containing the points of the line, a new list of FormationPoint elements is created, containing a point of the line and a Boolean to indicate if it can be used for expansions, this was still useful for debugging purposes by deactivating points and showing them with the visual debugger implemented.
- **FormationTools:** Manages more general methods which could be used by other line distributor algorithms, a static class which does not require to be instantiated and is directly used by the line distributors to use those methods.

7.2.2. Main function

It is located in the *UnitsFormationPositionGenerator* class and it is called *GeneratePosition()*. This class is integrated in the videogame in the method *ApplyFormation()* of *FormationBoardManager*, due to the functionality of this method, *GeneratePosition()* returns a double nested list of Vector3 elements, which represents a list of positions of elements for each line, while the problem is solved in two spatial dimensions, *ApplyFormation()* needs a function

which returns a list with three dimensional points, however the third dimension is left with value 0 and only the first two contain the solution. In the case of the parameters required:

- **List<List<List<Vector2>>> lines**: Contains all the points of all the lines drawn by a user. As the videogame has different types of units, and the elements used by each line is affected by the type of units, the list also makes that distinction. Therefore, the first list is the type of unit, the second list is the lines for that type of unit and the third list is the points for each line.
- **FormationBoardManager.FormationBoardDrawingData.DrawingScope drawingScope**: As the lines are differentiated by the type of unit, this indicates the current type of unit which is being calculated.
- **int elementsCount**: Number of elements of the type of unit selected, indicating how many elements must be inserted in the lines drawn by the user.
- **int separationDistanceElements**: Each type of unit occupies a different amount of space, which is the condition that defines how many elements fit in a line and is used to calculate the expansion of the original line.

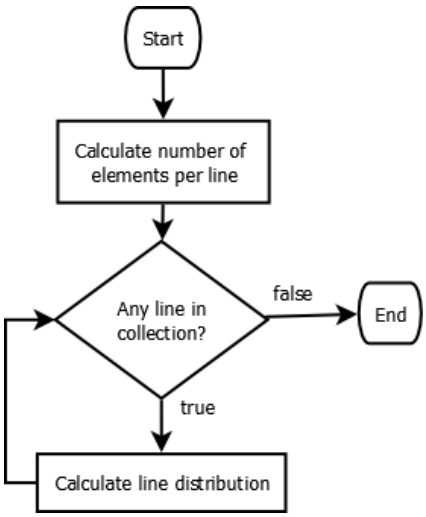


Figure 18 Flowchart of the main function

Once all the parameters are given, the main function follows the process shown in Figure 18, it begins by calculating the number of elements per line, for that it starts by assigning an element for each line only having one point, as that is treated as a special case in which the user only wants to insert one element, and after doing all the single element cases, the rest are spread through the other lines proportionately depending on the length of each line.

Then, after knowing how many elements each line has, a line distribution algorithm is passed through each line to find the position of the elements and finally the list is returned by the function.

Originally the main function was also supposed to detect closed lines and separate them from the open ones to be solved with a different methodology, however as the feature of detecting close/open lines was removed of the project, this part was not developed either.

The integration is made by calling this function in the *ApplyFormation()* method of *FormationBoardManager* and passing all the parameters mentioned, the call is made one time for each type of unit, and in the current version of the videogame there is a total of six types of

unit, therefore that is how many times the function is called every time the user draws a line, this could be improved in the future by only recalculating the lines for the type of unit which the user added a line.

7.2.3. Line distribution

While *UnitsFormationPositionGenerator* is the controller who manages which line distributor to use and makes the required operations to prepare the lines prior finding the positions, the line distribution algorithms are where most of the logic of the problem is featured, as explained before, there is an interface system *LineDistribution* to standardize the calls to those algorithms, however as the only algorithm implemented in this project is the *around the line* solution, this part focus on explaining how that solution in particular works.

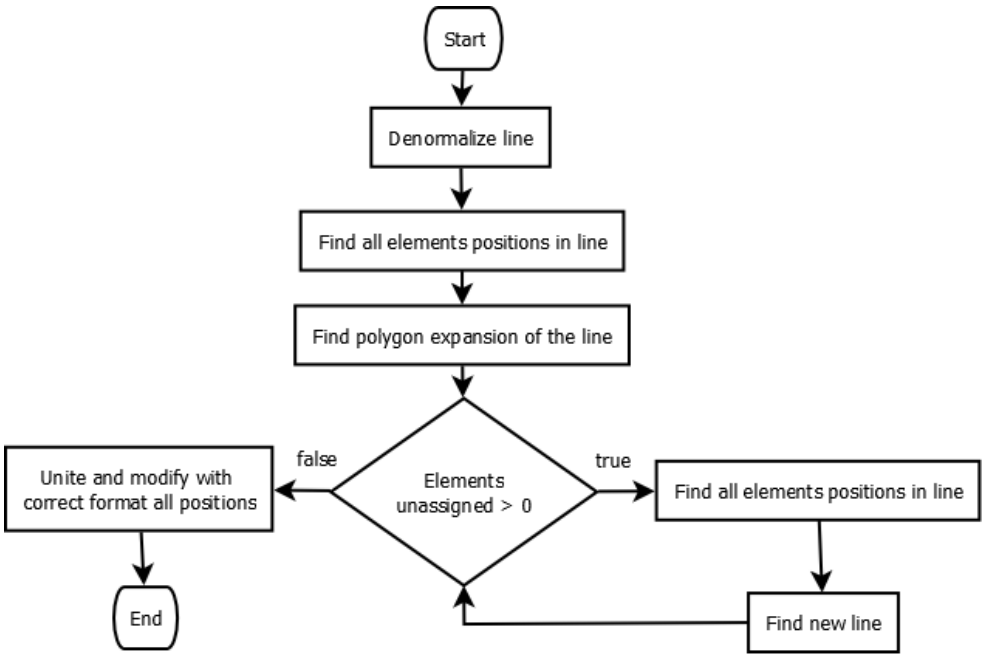


Figure 19 Flowchart of around the line, line distribution method

To begin with, there is a main loop, shown in Figure 19 and found in the *Run()* method. As the line is given with normalized values, the first thing is to denormalize the line to get the raw global positional values of the points making the line, there is two reasons to do this: first, the distance between elements given in the parameters uses global values instead of normalized ones, and secondly, the results of the operations must be returned also with global values, the process can also be done by normalizing the parameter and denormalizing everything at the end, but it is easier this way.

After that, the first line is filled using the method *FindAllPositionsInLine()* (this method is also later used to fill the expansions of the line), to do that it uses the *evenly scattered* method defined in *Designing solutions*. Which works by doing the following steps:

1. **Find the total length of the line**: Calculating the distance between two points p_1 and p_2 as the magnitude of the vector $\overrightarrow{p_2 - p_1}$, make a loop through the points of the line calculating the distance between p_i and p_{i+1} with i being the iteration through the list, while doing this, add all the values to get the total length.
2. **Calculate the capacity and scattering:** The capacity is equal to $\frac{total\ length}{distance\ between\ elements}$, while the scattering distance is $\frac{\text{capacity}}{\text{trunc(capacity)} \cdot distance\ between\ elements}$. This is necessary to know how many elements fill the line and the distance separating them.
3. **Starting point:** Define a starting point p_i, in this case pointing at the beginning of the list of points of the line, also define a distance travelled beginning on 0.
4. **Check and add position:** From the current distance travelled, find the relation with where the pointer is and add the position interpolating in the edge (p_i, p_{i-1}), before adding the position, check if there are other positions already generated which overlap with the new one, rejecting the position in that case as it is already occupied by an element.
5. **Update pointer:** If the current distance travelled surpasses the longitude of the current edge, update the pointer as many times as necessary by increasing $i = i + 1$ and decreasing the current distance travelled by the longitude of the current edge.
6. **Repeat the process**: From point 4 until all the positions the line is capable of containing are added.
7. **Remove unnecessary positions:** As there can be more positions generated than elements to be assigned, if that is the case, remove evenly the positions until there are the same number of positions than elements.

This works for all the cases except when only one element fits in the line, in that case add the element in the middle of the list. It could also be done by calculating half the distance of the total length of the line and iterating until reaching that point for a more accurate solution, however time is saved by using the first implementation for an acceptable performance given the points of the line are approximately equidistant from the adjacent.

Once the original line is filled, if there is still elements which has not been assigned to the line, it is necessary to define the expansion of the line, and that is done by the method *FindPolygonExpansionAroundLine()*, which makes different steps to define a polygon surrounding the original line, Figure 20 shows them visually:

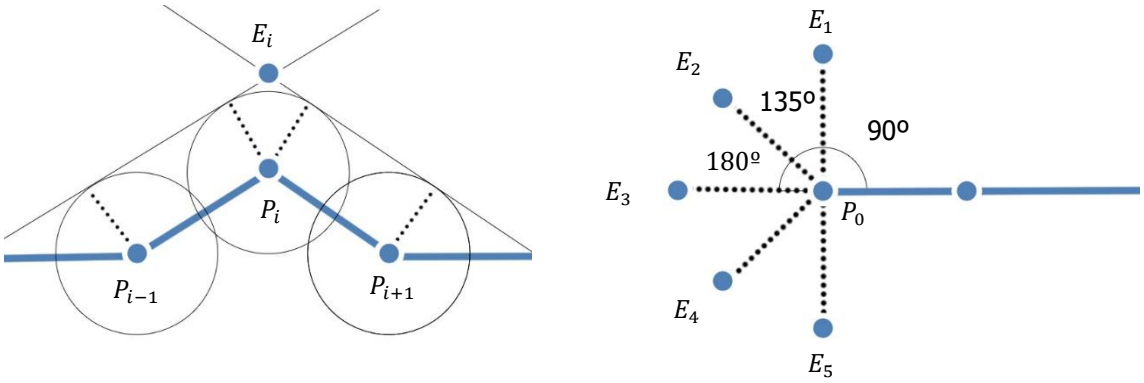


Figure 20 Graphs showing how the line is expanded, on the left shows the parallel lines to original line solution, on the right is the solution for corners, in these cases P represent a point of the original line and E the expanded points

1. **Parallel lines to original line:** Use the *circles tangent intersection* of *around the line*. Realized by a loop from beginning to end of the line but excluding both beginning and end. For each iteration, we have a point P_i and the adjacent P_{i-1} and P_{i+1}, from each of these points, throw an imaginary circle of radius equal to the distance between elements and find the tangents between the pairs of circles of $(P_{i-1},\ P_i)$ and $(P_i,\ P_{i+1})$, after that, calculate the intersection between those two tangents to find the new point of expansion E_i, the process is repeated by doing another loop of the line in the inverted order. This does not work for three aligned points, as the tangents are parallel and therefore the intersection does not exists, and although while they are strictly not necessary, those points are added using the *perpendicular* method, with in that case generates points in the same positions they should be to be parallel.
2. **Corners:** Use something similar to *perpendicular* of *around the line*, either in the corner P_0 or P_n, with n being the last point of the line, and with the help of their adjacent points P_1 and P_{n-1} respectively, the normal of the vectors formed by $\overrightarrow{P_0 - P_1}$ and $\overrightarrow{P_n - P_{n-1}}$ is calculated and using that as the direction and the distance between elements as the distance, gives the first point of the corner E_1, the same process can be repeated with the negative value of the direction to find the point at the other side E_5, then add the rest of points in between. To find the one in the middle, the direction to use would be either $\overrightarrow{P_0 - P_1}$ or $\overrightarrow{P_n - P_{n-1}}$ and getting E_3, and with those three points a corner is already defined, the only problem is that only those the points are at the distance between elements defined, but the ones between $\overrightarrow{E_1 E_3}$ and $\overrightarrow{E_3 E_5}$ are at less than that distance, to improve this solution a little more, two extra points are added between $E_1 - E_3$ and $E_3 - E_5$, for those the

direction is calculated adding the pair of directions used to calculate the correspondent expansion points and normalizing that value, finding E_2 and E_4.

Aside from these, there are more steps defined in exceptional cases. After finding the polygon expansion around the line, a loop is done, using *FindAllPositionsInLine()* to fill the last line calculated and *FindNewLine()* to find the next expansion of the line, using a similar logic to the one used to finding the polygon expansion, however in this case the expansion is always done from the first initial polygon found, and the value of distance between elements used to calculate the distance between the original line and the new polygon is multiplied by the number of iterations done to find the next lines. Also, as the polygon is already enclosed, it is not necessary to use the method for corners and only the method for parallel lines is necessary.

7.3. Exceptional cases

As it was not possible to find a generalized solution to this problem of expanding the line, there are cases in which the algorithm of tangents chosen for it does not work properly, in those cases, some extra logic is added to solve it. It could also be solved by limiting the possible drawings which the user can draw, but that would affect the gameplay experience and limit the choice for the developers, so it was not desirable and that is why it has been done. The main cases identified are the following ones:

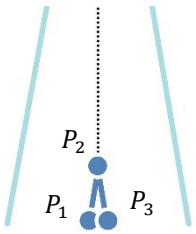


Figure 21 Example of wide angle, P_2 being too wide

- **Wide angles:** As the tangent solution generates parallel lines to the original, if the angle generated by three points is very wide, the distance of the expanded points grows towards infinity, therefore, while the expanded line still maintains the right shape, it is inviable from a practical point of view, since the point could be so further away from the original line that it could not be used as a unit formation.

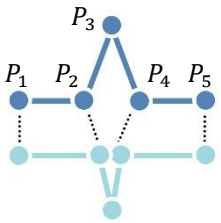


Figure 22 Example of narrow angle, P_3 being too narrow

- **Narrow angles:** Similar to wide angles, though this case is particularly noticeable when the three points have a very narrow angle and the ones surrounding it become wider, going from a convex angle to a concave one, since the narrow angle will generate an expanded point far away, while the others will stick closer to the line, totally misrepresenting the shape of the expanded line.

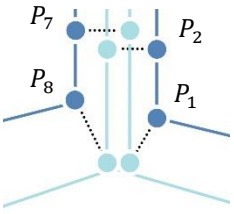


Figure 23 Example of narrow passage, P_2 and P_7 are on the opposite side

- **Narrow passages:** If there is a point where two edges are too close to each other and with opposite directions, it is possible to have a situation in which the distance between elements used to define the parallel lines is greater than the sum of the distance between both edges, in those cases the expanded points swap positions, as the expanded point of one edge is closer to the other edge instead of with itself, causing the new polygon to break, especially in small loops and the more expanded lines are created.

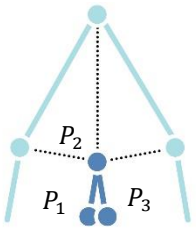


Figure 24 Solution for wide angles

For wide angles, the solution is to cap the distance the expanded point can be, for that a frontier angle is selected, in this case was 330º. After deciding for a frontier angle, it is possible to find the distance at which the expanded point is found in that case using trigonometry, which at the end gives the following formula:

$$distance = \frac{distance\ between\ elements}{\cos\left(\frac{angle - 180º}{2}\right)}$$

The result of that distance for 330º is a multiplier of distance between elements of 3.86. Also to make sure that the expanded line keeps parallel to the original one up to the point from where it is being created, using the logic of tangent and having the three points P_1, P_2, P_3 as defined before to generate the circles and the tangents, the tangent points in the circle of P_2 created by the tangents between (P_1, P) and(P_3, P_2) are also added before and after the expanded point.

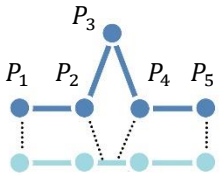


Figure 25 Solution for narrow angles

For the narrow angles, if there is enough points defining the line, it is possible to closely approximate the shape of the line in the corners made by those angles without the need of that point, therefore at a certain angle, that point is not even generated, for this project selected at 90º, as it starts becoming troublesome soon, even before reaching more extreme angles.

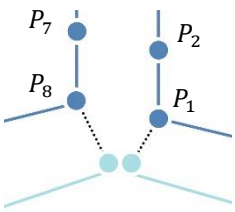


Figure 26 Solution for narrow passages

Finally for the narrow passages it is the most expensive solution, as it requires to add a loop to the generation of tangents but still falls in the O(n^2) category. For that, every time an expanded point using tangents is generated, starting from the original point, using the direction of the expanded point and the distance being twice the value of the distance between elements, a line is created and is compared which each of the edges of the original line, if the created line intersects with one of the edges of the line, the expanded point is not generated.

7.4. Recognizing open/close patterns

Originally this part of the project was planned to recognize when the user closes the beginning of the line with the end, of course that can be as simple as finding if there is an intersection between $\overrightarrow{P_0P_1}$ and $\overrightarrow{P_nP_{n-1}}$, however it is also necessary to consider the accuracy of the user, and therefore there could be many more cases, like that intersection being found between a different pair of edges, or the user not even closing the shape, but the initial and ending point being so close as to suggest that the line was closed. Then, the plan was to implement a different distribution in those cases by trying to fill the close shape first instead of trying to fill the positions closest to the line. However due to the complexity of the original problem this feature was discarded.

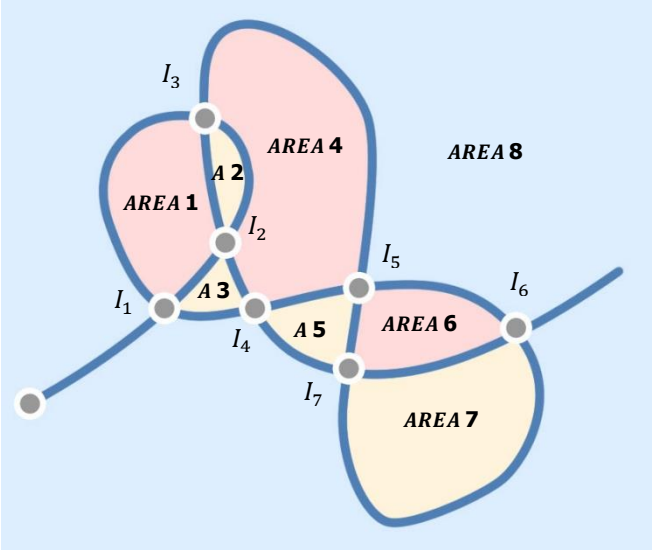


Figure 27 Closed areas of the line identified, the points of each area are the ones of the line surrounding it, making a total of 8 independent lines, and each I represent the intersections of the line

But something similar was studied for its use in the project, in this case the idea is that given a line, try to detect all the closed lines to expand them independently of the other ones found in the line, the idea about this is that by dividing the original line in simpler independent blocks, it would be possible to remove many exception cases, as it removes all the intersections of the original line. However, it was proven that this would be too costly for the implementation of this videogame. For that reason, this part of the implementation is going to explain the solution to recognizing close lines which was designed for this part and then it explains why this could not be used in the current project.

What this part tried to do can be more easily seen visually in Figure 27, having that line as the example, try to find an algorithm which is capable of returning a list of points for each of the closed areas identified in the line, so that the points are contained within the area.

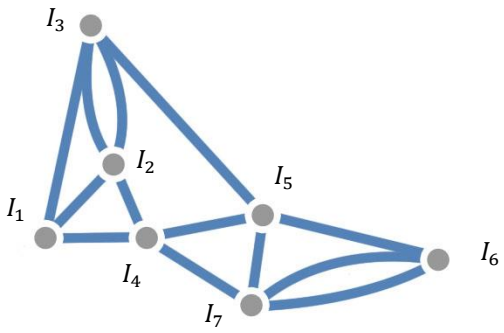


Figure 28 Graph of the example studied

The first observation about this problem is that it is not possible to create a closed area without an intersection, therefore the intersections are the key piece of information to find define those areas, and by removing all the points of the line and just drawing the intersections and the links between them, it is possible to find a simplified version of this problem, as shown in Figure 28.

By reducing the problem to its most basic shape, it is easy to identify it as a graph, therefore making it possible to use graph theory to solve this problem. And there is something very closely related to this, which is *cycles*, with a cycle being defined as a trail in which the only vertices which repeat are the first and the last one. That definition does not consider the cases in which there is an edge inside the trail, however for case there is another more precise definition, which are the *chordless cycles,* they are also cycles, but there cannot be two vertices from the cycle which are connected by an edge which is not part of the cycle, and there are solutions to this problem already designed. Of course, this solution does not consider the initial and ending section of the line, but those can be treated as an exceptional case.

After doing research, the main methodology to solve this problem is to make some type of tree search, some of them are the following ones:

- The first solution is a stackoverflow answer (23), which defined a tree search, starting from each vertex and exploring all the surrounding vertexes, and depending whether they directly connect back to one of the first vertexes a chordless cycle is identified, this process is repeating until all the combinations are found, though it a O(n^3).
- Another one explores a possible solution for directed graphs (24), and while our case is not a directed graph, maybe it could have use due to the line having a direction, however that is not the case and the estimations shows that his implementation is at least O(n^4).

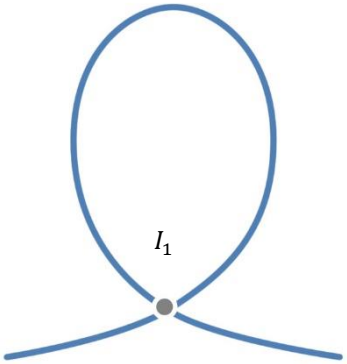


Figure 29 Example of loop

- Finally, the best one found (25), which also reviews other implementations, utilizes a Depth-first search strategy, starting from the bottom by defining all the possible vertex triplets (combinations of three vertexes) and adding surrounding vertexes until finding all the chordless cycles, in this case the complexity is of O(n^3), therefore while still costly, this would be the best algorithm which could be used, as it still enters within the requirements of the project.

However, there is a small problem, all these solutions only work for simple graphs, which means that they cannot have loops or repeated edged, and the line can have loops, as it is as simple drawing as drawing a line with intersects itself with no other intersection being made, as shown in Figure 29, and also there is cases of repeated edges, such as the ones found between (i_2, i_3) and (i_5, i_6) in the example case of Figure 28.

But those cases can be solved individually, for loops it is necessary to follow the line in order and detecting if at some point an intersection is crossed and if that intersection is crossed again before reaching any other intersections, it is theoretically possible that a loop can be repeated more than once, but the process works the same. For the repeated edges it only needed to follow the order in which the intersections have been found in the line (which is the same order in which the line is defined) and whenever two adjacent intersections (I_x, I_y) are found, if that pair of intersections is found again that means there is a chordless cycle, however it can become more complex if the repeated edges happen more than twice, while unlikely practically, it could still be possible to do theoretically, and in that case it becomes more complicated because once all repeated trails have been found of the pair (I_x, I_y), it is necessary to identify the adjacent trail for each one of them.

After that all the chordless cycles are found, and therefore all the segments of closed lines can be identified, however, one problem with this is that the list of the line must be ordered in a particular way, and the solution of the chordless cycle does not indication the direction in which the points must be stored. Considering that the side of the line which is expanded using the algorithms of the solution is the one perpendicular to the points p_i and p_{i+1}, using a similar computational geometry solution as the one presented for the point-in-polygon problem, it is possible to detect whether it is necessary to reverse the order of the list or not. For that, choose two adjacent arbitrary points of the closed line p_i and p_{i+1}, throw a perpendicular line and detect all the times it intersects with the closed line, for an even number the perpendicular

is inside the closed line and therefore no change is needed, for pair however the order must be reversed, as show in Figure 30.

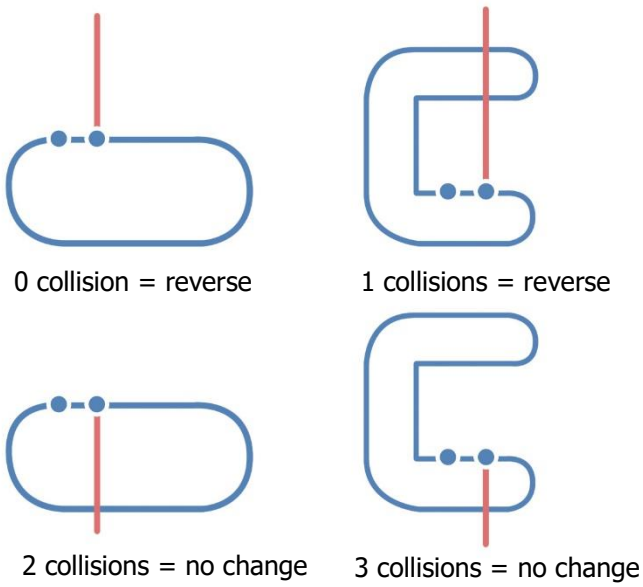


Figure 30 Examples both of convex and concave polygons showing how to detect whether to reverse the list of the line or not, the red line being the perpendicular thrown from the points p_i *and* p_{i+1}

After that, all the chordless cycles are solved and therefore all the closed shapes are correctly identified, however there is still an area from Figure 27 that has not been considered yet, and that is Area 8, which is the area around the line, as it is also necessary to identify the points of the line which allows to expand it outwards. The solution is to take an extreme point of the line, which for example could be the one with the highest or the lowest value of one of its dimensions and following the line in the usual direction, however, when finding an intersection, and considering the direction of the last two points added to this list, figure out the angle this one does in relation with the other lines which crosses the intersection and select the one with the lowest or highest value of the angle, in a more simplified way, when finding an intersection, continue through the leftmost or rightmost line, aside from that if a the starting or ending point is reached, reverse the order of the search and keep going, repeat the process until going back to the initial point.

So, there is a plausible solution to the problem, the question then is why it was not implemented, and that is for many reasons. First, simplicity of the solution of the project is encouraged, and this is quite complicated both to implement and to maintain, and it would make it really hard to make changes to the system as it affects to how the original solution

works, and therefore in case of this being removed, it would require more effort than just deleting it.

Aside from that it is still costly on time, as shown with the solutions of chordless cycles, it is possible to reduce the time complexity to O(n^2), however that is not the only part of the closed lines solutions which requires O(n^2), so while it is inside the limits of this project, it is still not desirable. Finally there is a property about intersections which can be problematic, it is possible to have more intersections than points defining the line, it is not the common cases, but it is still possible, and if that happens it can affect greatly the performance of the project, just to show an extreme example of this happening, see Figure 31.

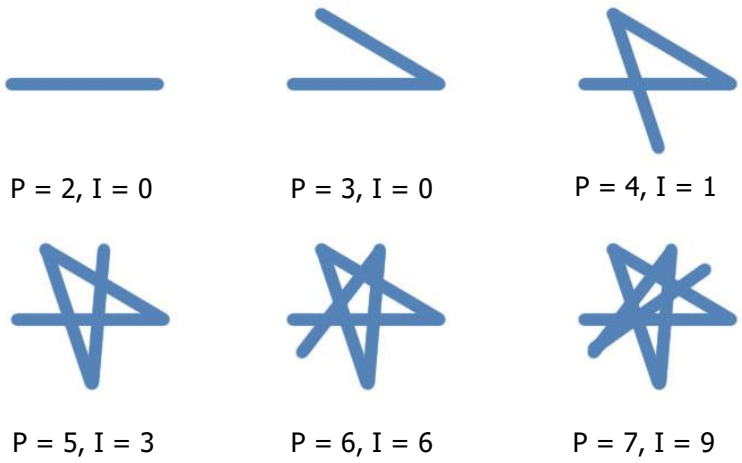


Figure 31 Example of line having more intersections than points of the line, P is the number of points and I is the number of intersections

8. Results

Overall the project has been a success, as far as it has been tested, the solution works, there has not been extreme cases observed as the ones which have been found are fixed and the response time in the videogame observed is instant, therefore accomplishing what was expected at the beginning.

To see it better below shows the results in the videogame, different combinations and the different things which had to be considered to make it work for all the scenarios it must support, most of the examples are done with 500 units, which is far from what the videogame will have to handle, but what was requested for it to manage.

Some images show the units making the formation that has been drawn, while in others it also shows the debug system created to find issues while developing by showing the lines which are created. In that case the blue line is the original line drawn by the user, and each of the green lines represents the expanded lines generated. Aside from that each block represents one of the points of each line, the white blocks represent the start and the black ones represent the end.

8.1.1. Simple cases

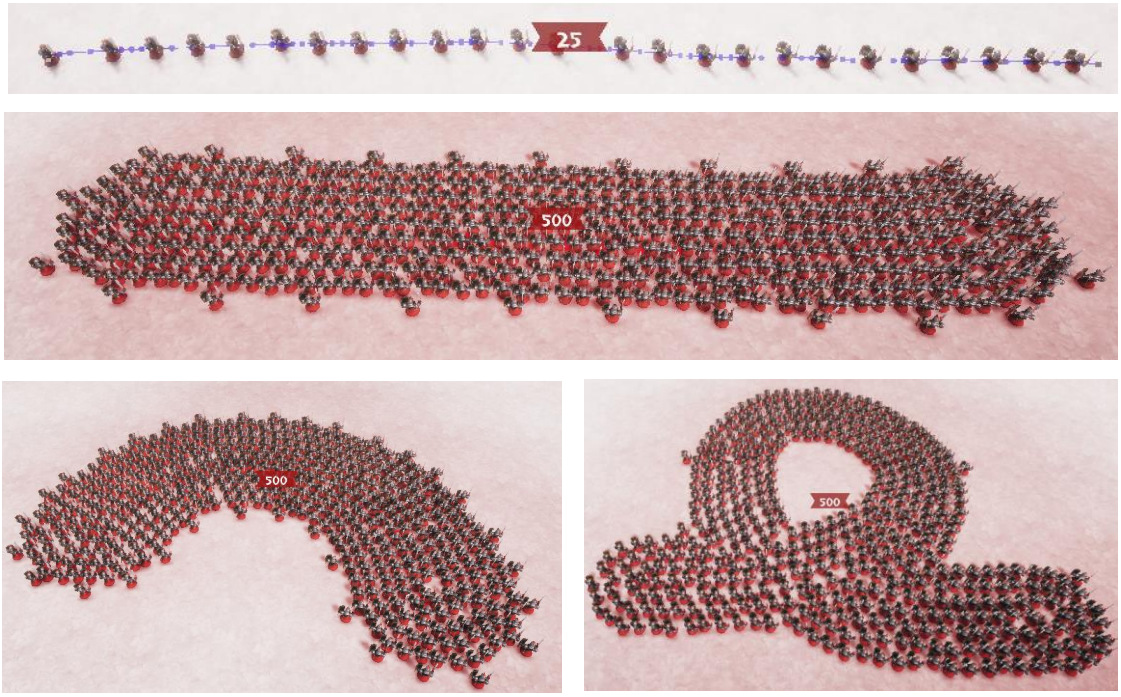


Figure 32 Some basic examples, showing a single line, a straight line, a curved one and a loop going from the simplest to a more complex case

The simplest case to do is a single line, as it does not require to create any expansion of the line. The next one would be a straight line, as the parallel extended lines are the same than the original one but displaced to both sides. Then, the moment Some curvature is added, things get a bit more complicated, as one side is going to grow in size and the other one is going to shrink, with the possibility of shrinking so much that there is no more space to continue, and finally there are the intersections, where the lines crosses and therefore the expansions also have collisions, in that situation it is necessary to analyse whether to control it after doing the lines by checking if other units are occupying a space when assigning a new one, or removing those collisions when defining the lines, which is what the solution of recognizing open/close patterns attempted to do.

Luckily as seen in Figure 32, the approach of fixing it after doing the lines works fairly well, and the only noticed problem is that there are sometimes holes inside not occupied by units. The last line also contains fewer units than the rest, but that is because the last line is not going to have always enough units to be filled, therefore they are evenly distributed around by design. Another possible solution to that would be to recalculate all the positions again but distributing the units from inside to the last line so that all the lines are equally filled.

8.1.2. Unit size

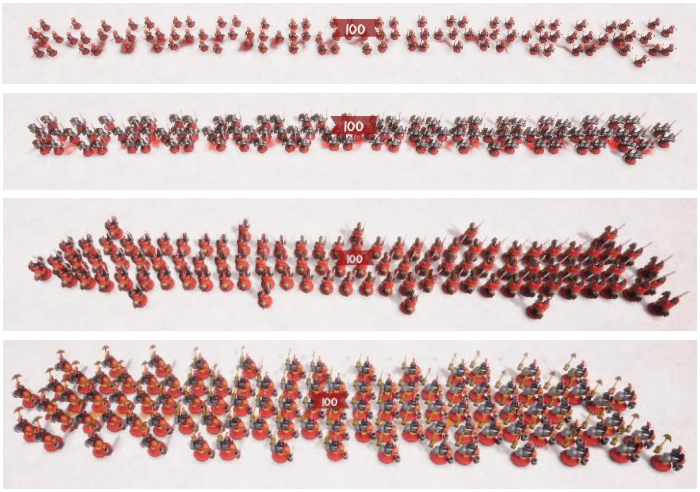


Figure 33 Example showing different units on same formation but spreading differently depending on the distance assigned to them, from top down, there are the archers, infantry, cavalry and heroes

Of course the algorithm must be capable to adapt to the size of the unit (or the space assigned for them to spread), something which can be seen in Figure 33, also as they occupy more space, it can be seen in the picture that due to size, the units at the bottom have to extend one more row compared to the first two for a line of similar length. There has not been as much testing of this feature, but from the tests done, the algorithm seems to work properly.

8.1.3. Exceptional cases

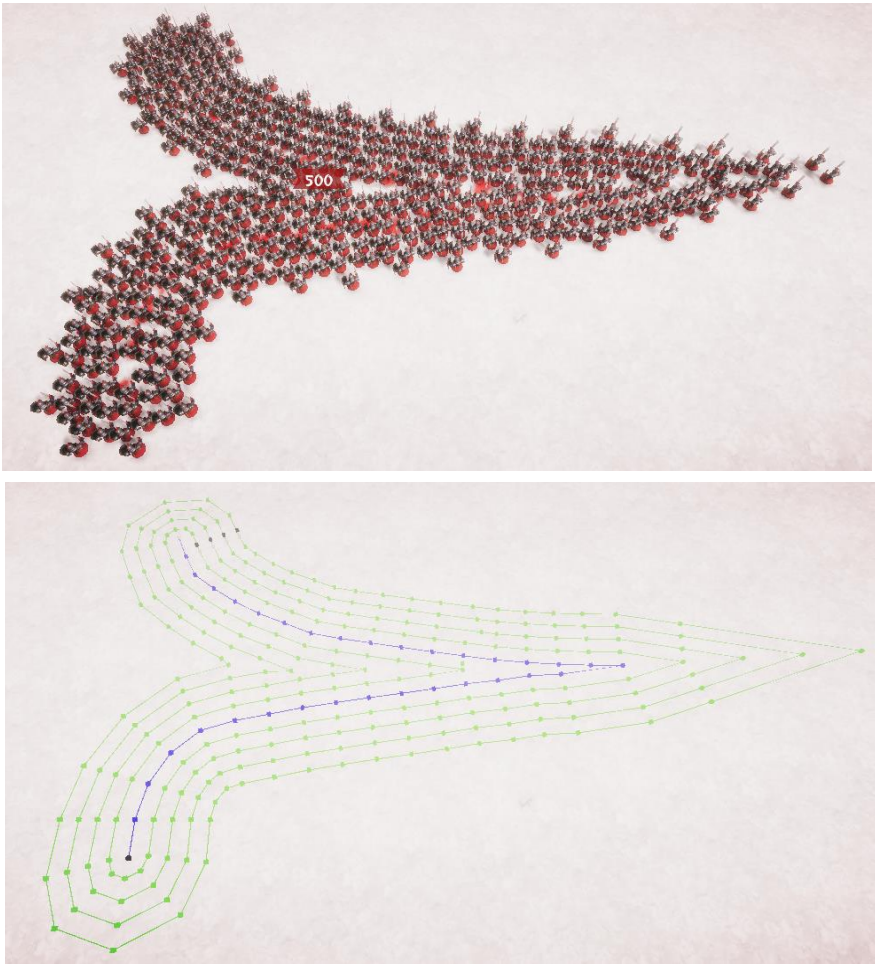


Figure 34 Example of wide angle, narrow angle, and narrow passage

As seen in *implementation*, there are cases which are particularly complicated for the solution of tangents, Figure 34 is a good example showing the three of them at once due to the very extreme change of direction which the original line does in the middle and how the line is very close in each side to the other. In the wide angle, the distance is limited so that it does not extends further away, in the inside the narrow angle being so extreme is not used, and the points close to the angle are not used either to generate points on the inside of the curve.

In some cases the line shows holes in the inside, as the closest points of the lines which can extend are too far away from the point where it should actually extend, but it is not very noticeable and even then it can be fixed by just increasing the amount of points representing the lines.

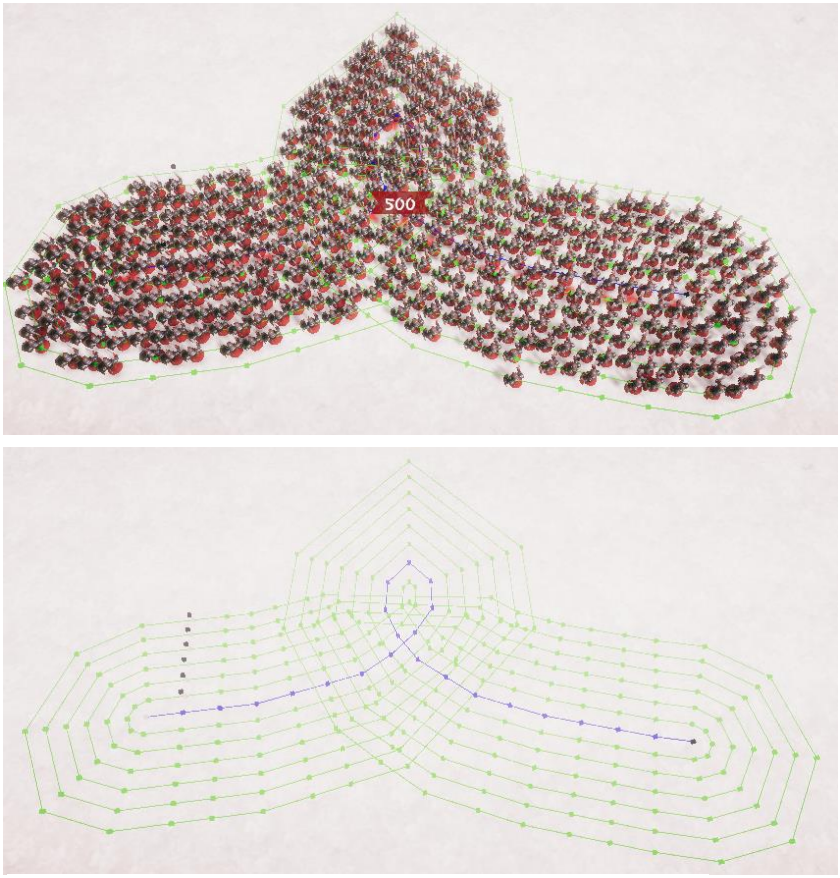


Figure 35 Example of loop

To visually see what happens in a loop check Figure 35. As mentioned before the problem with loops, which also can happen in other situations are the intersections, but also creating a closed shape inside which is going to have a limit in regards to how much it can grow, and that can be seen there, as there is only one extended line generated in the inside.

However, something odd about this case is those extended lines crossing inside the loop, but that is the result expected, that is made by the solution for the exceptional case of narrow passages as it should. As new lines are extended, no point is generated inside the loop, and in the current version the line does not break when this happens, as there can be situations in which is necessary to fill that gap, therefore those horizontal lines happen due to the algorithm connecting the last point in the left before reaching the loop and the first point in the right after the loop, while it does not look good in the debugger, it works fine in practice when filling it with units, as those lines are inside of the shape of the expansion.

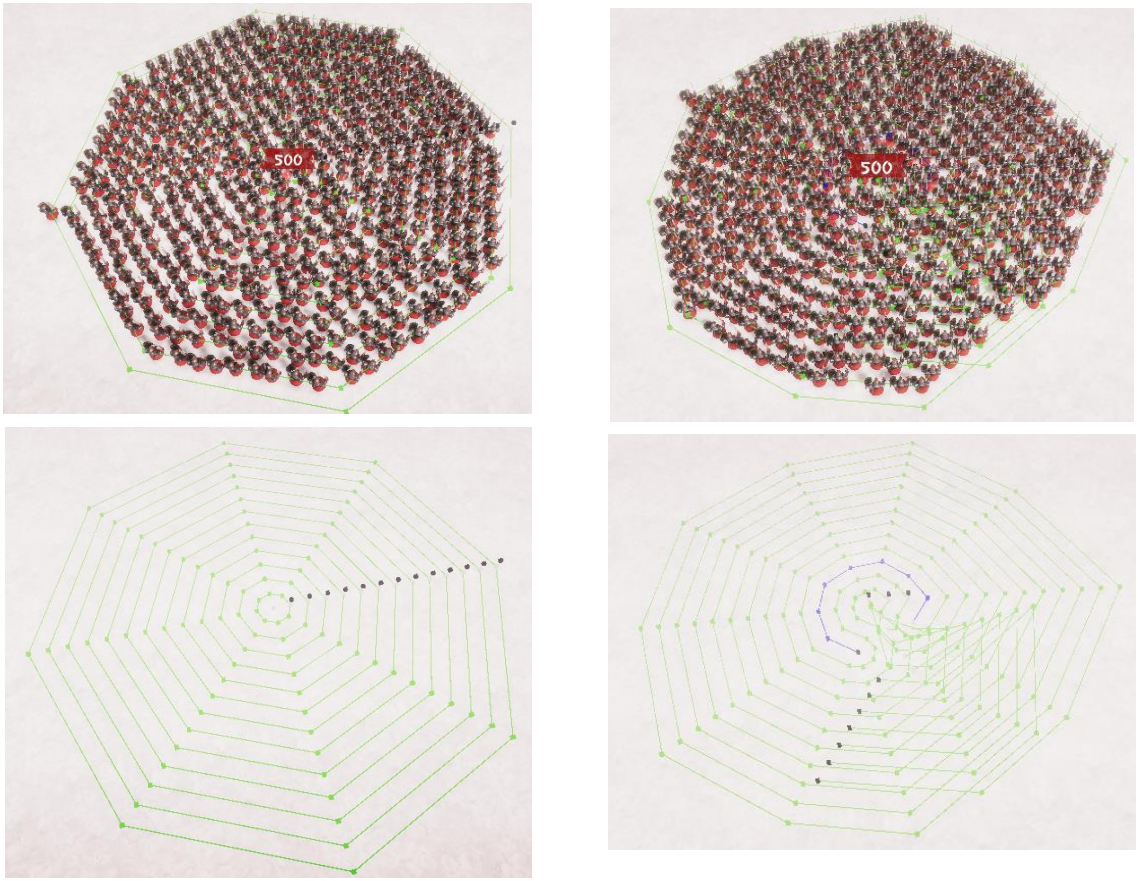


Figure 36 Examples showing the solution for one a line made by a single point in the left, and a small circle in the right

While there must be only one unit in the line if there is only one point defining it, there exists a couple of situations in which is necessary for it to be able to fill the point with more units. If the point is the first line drawn, then that is the only line which can be filled, and therefore the point must be capable of containing all the units, if there are also more points generated but there are more units than points, then one of the points must be responsible of containing those units. The solution for that is just creating the initial polygon with 8 points around, that is why the line generation looks so evenly distributed in the shape of an octagon in Figure 36.

The other case shown in Figure 36 is a very small unclosed circle, that is probably the most problematic case found, and as seen in the debug, the lines become a mess in the part where the circle opens, however the results when placing the units do not show that, as the lines generated are all inside the overall shape made by the expansion.

8.1.4. Line complexity

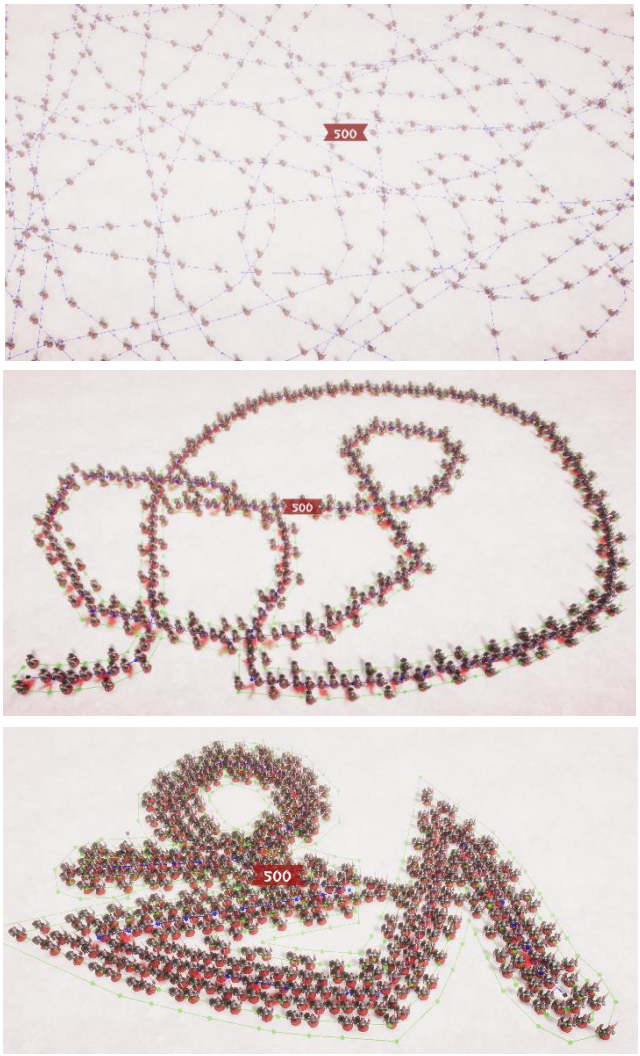


Figure 37 Example showing complex lines, from longer at the top, to shorter at the bottom

Finally some extreme cases to show that the algorithm still works even when things get more complicated as show in Figure 37, the first case is a very long line, in which all the elements fit, and therefore it is not necessary to create an expansion of the line, something which actually is necessary for the other cases, and still it works as intended.

One limitation about this however is that it can reach a point where the delay of calculating it is noticeable, for example the first picture are around 30 seconds of drawing a very long line, and after finishing it, there was a small delay of 1 second before showing the results while freezing the game (as the current version is sequential instead of working asynchronously) Still, that is a very good result given the complexity of the case, and is something which could be easily controlled by limiting to some degree the longitude of the lines drawn.

9. Conclusions

This problem is more complicated than it seems at first sight, while all the algorithm is doing is spreading elements in a line, things get complicated when expanding it and considering that the user is capable of drawing anything they want in one trace. That is something which is reflected in the methodology, as features took longer to make or even cancelled due to not having enough time for them, luckily at the end it was possible to make a solution which works, however at many points of the project it did not seem like that was going to be the case. As there is no point of reference in this subject, everything has been about figuring out possible geometrical solutions and experimentation.

Probably the biggest mistake of this project was using so much time trying to solve the problem of recognizing open/close patterns, while at the end it was possible to design a solution, a lot of time was invested on it, and at the end it was too complicated and there were so many potential problems with it that it was not worth attempting to implement. As a core idea of the project was to keep things simple, therefore the investigation should have stopped sooner, and more time could have been invested in improving the current version.

The algorithm still needs some polishing to make sure it works for all the cases, and there are still some things which could be improved, but even when testing it in difficult scenarios the algorithm have worked so far and it can be used in its current state, therefore, while there is still work to do and some planned things could not be done for this project, we consider it a success.

While the videogame could still work without this feature, just by using any of the other methodologies studied in the state of the art, we think there is value in this method and it should help the videogame to stand out over other videogames of the RTS genre, as we have not found cases using the method suggested in this document, and even the closest one, which is Tactile Wars, ceased in popularity after many years of being on store and being an online mobile videogame, there should not be other competition in regards to the feature developed.

9.1. Future work

As mentioned, there are some features which were not developed and the code needs some polishing, therefore here is a list of things to do after what was achieved for now:

- **Better generation of unit positions:** One problem shown in the results is that sometimes there are holes inside the formation, this could be fixed by making a better generation of the unit positions. A possible solution could be to displace all the units before the collision. Considering there is a gap not big enough for the unit, by increasing how separated the units before the collision are until they reach that position it could be solved, units in that line would be a little more spread, but it would be less noticeable than leaving the hole as it is.
- **Feature of closed shapes:** While the result of the investigation done for recognizing open/close patterns found an unviable solution for that, this is more about the original idea, of recognizing when the user has drawn a close figure, in that case there would be only one polygon and the lines not contained within could be removed. To recognize if it is just one polygon it is only necessary to check how many intersections there are in the line, and there is only one polygon when there is exactly one intersection, after that it would be possible to apply a expansion/shrinking algorithm for polygons, something which already exists.
- **Asynchronous:** Another problem mentioned in the results is that the screen can freeze for really long lines, this could be solved by making the feature asynchronous to the rest of the logic of the videogame, so that the algorithm can be executed in a thread and give the solution whenever it is ready, and meanwhile the game still works, not being stopped by the calculations.
- **Only calculate one line:** The current version calculates all the lines of all the types of units every time the user draws a new line, however this is not necessary, if the user draws a line, it only affects to the other lines drawn for that type of unit. Not only that, but also it is not necessary to recalculate the expansions of the other lines, because the more lines are generated, the less units there are in each line, therefore there is already enough space calculated to fit the units in the other lines after adding a new one, and the only thing which actually needs to be calculated is where to put those units in regards to the positions generated for each line, which should be cheaper.
- **Optimization:** While the algorithm works fine, there are still some expensive calculations which are being made and that are not necessary, like using vector magnitudes instead of the square root magnitude, that is something which could increase a lot the performance of the algorithm.

- **Refactoring and documentation:** Due to the limitation in time it was not possible to make all the code clean and easy to use, therefore it would be helpful to add all the documentation which is missing and fix to code to make it easier to manage and more easy to read.

10. Bibliography

1. *Strategy in games or strategy games: Dictionary and encyclopaedic definitions for game studies.* Dor, Simon. s.l. : Game Studies, April 2018, Vol. 18.

2. Real-Time Strategy (RTS). *Techopedia.* [Online] 21 April 2015. [Cited: 30 July 2020.] https://www.techopedia.com/definition/1923/real-time-strategy-rts.

3. Entertainment Software Association. *2019 Essential facts about the computer and video game industry.* 2019. p. 21.

4. Clash Royale Review – Should You Play It? *clapway.* [Online] [Cited: 30 July 2020.] https://clapway.com/2016/03/15/clash-royale-review-play/.

5. Castle, Louis. *How Command & Conquer: Tiberian Sun Solved Pathfinding | War Stories | Ars Technica.* [interv.] Ars Technica. 26 February 2019.

6. TheViper. TheViper TUTORIAL: How to micro Ranged Units vs Mangonels. *Youtube.* [Online] 17 September 2016. [Cited: 30 July 2020.] https://www.youtube.com/watch?v=v6zDm7RzHNc.

7. NeOmega. Fantasy RTS in development. Draw Formations, Airships and soldier formations. *Youtube.* [Online] 5 December 2018. [Cited: 30 July 2020.] https://www.youtube.com/watch?v=4dWCDWLwg1s.

8. Orbital Nine Games. Brain It On! - Physics Puzzles. *Google Play.* [Online] 14 July 2020. [Cited: 3 August 2020.] https://play.google.com/store/apps/details?id=com.orbital.brainiton&hl=en.

9. ANKAMA GAMES. Tactile Wars. *Google Play.* [Online] 28 June 2018. [Cited: 3 August 2020.] https://play.google.com/store/apps/details?id=com.ankama.tactilwar&hl=en.

10. Bad Review Games. Car Drawing Game. [Online] 7 July 2020. [Cited: 3 August 2020.] https://play.google.com/store/apps/details?id=com.draw.car.fun3d.road.racing&hl=en.

11. Lacort Navarro, Jorge. Trello. *Strategy videogame.* [Online] [Cited: 10 September 2020.] https://trello.com/b/v8QeewVo/strategy-videogame.

12. Marina. Factorial. *El coste de un trabajador para la empresa [+fórmula].* [Online] 4 March 2020. [Cited: 10 September 2020.] https://factorialhr.es/blog/coste-empresa-trabajador/.

13. Indeed. *Salarios para empleos de Analista programador/a en España.* [Online] [Cited: 10 September 2020.] https://es.indeed.com/salaries/analista-programador-Salaries?period=monthly.

14. Indeed. *Salarios para empleos de Programador/a junior en España.* [Online] [Cited: 10 September 2020.] https://es.indeed.com/salaries/programador-junior-Salaries?period=monthly.

15. Rutherford, Sam. Laptop. *Asus ROG GL552 Review.* [Online] 25 March 2016. [Cited: 10 September 2020.] https://www.laptopmag.com/reviews/laptops/asus-rog-gl552.

16. Windows. *Windows 10 Home (descarga).* [Online] [Cited: 10 September 2020.] https://www.microsoft.com/es-es/p/windows-10-home/d76qx4bznwk4?activetab=pivot:overviewtab.

17. Microsoft. *Microsoft 365 Personal.* [Online] [Cited: 10 September 2020.] https://www.microsoft.com/es-es/microsoft-365/p/microsoft-365-personal/CFQ7TTC0K5BF/007R?source=googleshopping&ef_id=Cj0KCQjw0Mb3BRCaARIsAPSNGpU4Ie4ELuyYt4Uif6UV4m_GJmV09Nfp7Gq88sS_vmEiDPD9ctzivIIaArTdEALw_wcB%3aG%3as&OCID=AID2000751_SEM_xyMqANrA&MarinID=.

18. Unity. *Choose the plan that is right for you.* [Online] [Cited: 10 September 2020.] https://store.unity.com/compare-plans.

19. Regus. *UBICACIONES CON ESPACIOS DE OFICINA EN ZARAGOZA.* [Online] [Cited: 10 September 2020.] https://www.regus.es/office-space/spain/zaragoza.

20. Núñez, Andrea. TICbeat. *Cuánto pagas por el agua según la Comunidad Autónoma en la que vivas.* [Online] 16 June 2020. [Cited: 10 September 2020.] https://www.ticbeat.com/empresa-b2b/cuanto-pagas-por-el-agua-segun-la-comunidad-autonoma-en-la-que-vivas/.

21. Podo. *Factura media de luz en un hogar de España ¿Cuánto se gasta de luz al mes?* [Online] [Cited: 10 September 2020.] https://www.mipodo.com/blog/ahorro/factura-media-luz-hogar-espana/.

22. Jazztel. *Fibra Jazztel 100Mb.* [Online] [Cited: 10 September 2020.] https://www.mijazztel.com/internet_fibra/fibra_50?tsource=jazgen_004_gmasm_frases_n2_principal_oferta_10769773567&distribuidor=google&gclid=CjwKCAjwkun1BRAIEiwA2mJRWSyMkq8LiyZxt3sD19xXj9e9lTzX-6Jqe61Qd8J1ntl-1_eVQ4hBERoC5YgQAvD_BwE.

23. Smith, Eugene. stackoverflow. *Find all chordless cycles in an undirected graph.* [Online] 26 October 2010. [Cited: 28 August 2020.] https://stackoverflow.com/questions/4022662/find-all-chordless-cycles-in-an-undirected-graph/4028652.

24. Bisdorf, Raymond. orbilu. *Enumerating chordless circuits in directed graphs.* [Online] 28 January 2010. [Cited: 28 August 2020.] https://orbilu.uni.lu/bitstream/10993/23926/1/orbel24-2x2(1).pdf.

25. Silva Dias, Elisângela , Castonguay, Diane and Longo, Humberto. *Efficient Enumeration of All Chordless Cycles in Graphs.* Instituto de Informática, Universidade Federal de Goi´as. 2013.

Annex

Annex A

SPRINT REVIEW

Sprint 1 – Basic implementation

Time

- **Sprint date:** 21/07/2020 – 31/07/2020
- **Sprint finished:** 31/07/2020

There was no delay on this part.

Tasks

- **First draft of memory:** Everything went accordingly with this part, though there was more time invested in methodology than state of the art than what was expected.
- **Design base code:** Everything went accordingly with this part.
- **Integration to videogame project:** This part was supposed to be done at the end of the project, but due to some reframing of the project, it was necessary to do it at the start due to having to work directly in the project and not being able to do it separately. Also, this part took less time to do than predicted, but it was still reasonable.

Results

Everything went accordingly, there is no reason to change things for now.

Sprint 2 – Distribute elements in a line

Time

- **Sprint date:** 03/08/2020 – 11/08/2020
- **Sprint finished:** 13/08/2020

There was no delay on this part.

Tasks

- **Study possible distributions:** Everything went accordingly with this part.
- **Implement best distribution:** Implementing one of the distributions is complex enough that there was not enough time to finish one of them, and therefore the scope is going to be reduce to just that one which was implemented, not only that but a new task was added to the next sprint finish what was not finished before.
- **Second draft of memory:** More time was needed to do this first draft as it has all the design done for this project, requiring also to do many graphs for it, which also required to make some basic tools to make the graphs consistent.

Results

This sprint was not as successful as the first one, it went from requiring 7 days to be finished to 9, therefore some of the sprints later might need to be reduced in workload as the scope should be reduced as well. Main problem is that the tasks required to implement is more complex than expected.

Sprint 3 – Solve exceptional cases

Time

- **Sprint date:** 14/08/2020 – 20/08/2020
- **Sprint finished:** 21/08/2020

Tasks

- **Continue design and development of initial implementation:** At the end decided to use the tangents solution, mixed with the perpendicular one for parallel lines cases, everything went accordingly.
- **Study limits and exceptional cases:** Found multiple cases, specially when the line abruptly changes direction and managed to design some initial solutions, everything went accordingly.
- **Implement solution to limits:** Implementing the solutions is where the problems started to appear, since there were many more cases which were failing, and therefore they had to be fixed.
- **Test and fix limits:** This one came along with implementing the solutions, so all problems derived from it affected to this part.
- **Third draft of memory:** There was so much time invested on solving problems that there was no time to do this part.

Results

The problem definitely needs a reduction in scope, as it is much more complicated than it seemed at first, for now for the next part it'll actually continue with the part of recognizing close/open lines, however instead of its original use to make different formations depending on whether closed or open lines are detected, its functionality has been identified to solve many of the exceptional cases found.

Sprint 4 – Recognize close/open line

Time

- **Sprint date:** 24/08/2020 – 28/08/2020
- **Sprint finished:** 28/08/2020

Tasks

- **Define limits to recognition of closed line:** A lot more of time was invested trying to investigate a possible solution which would work for this case, however all this effort was proven pointless as all solution found were too costly in time and could probably induce to other problems. Therefore, this was a clear waste of time and the investigation should have been cut down much quicker.
- **Implement recognition of close/open line**: Not done due to limit in time.
- **Test and fix limits:** Not done due to limit in time.
- **Fourth draft of memory:** Not done due to limit in time.

Results

The problem as it has been tried to be solved, finding all the closed patterns in the line is too complicated to be solved for the game, therefore all the effort put into this was unnecessary and it should have been identified as that quicker to improve the current implementation. Which is what the rest of the project will be dedicated to do.

Sprint 5 – Fix problems

Time

- **Sprint date:** 02/09/2020 – 06/09/2020
- **Sprint finished:** 06/09/2020

Tasks

- **Fix problems of exceptional cases:** Finally, all the problems were fixed in time, everything went accordingly.
- **Draft of memory:** Everything went accordingly.

Results

Everything went accordingly in this part.

Annex B

REUNIÓN:	**DEFINIR OBJETIVOS**		
Fecha:	13/07/2020		
Hora comienzo:	23:00	**Hora finalización:**	01:00
Lugar:	Universidad San Jorge		
Elabora acta:	Jorge Lacort Navarro		
Convocados:	Antonio Iglesias Soria		

Orden del día / Acta

No.	**Asunto**	**Acuerdo**
1	**Disculpas por ausencia**	
2	**Aprobar última acta**	Primera acta
3	**Asuntos pendientes última acta**	
4	**Otros asuntos**	
5	Debatir los requerimientos del proyecto y posibles diseños iniciales	Pensar en las herramientas necesarias para el proyecto
		El proyecto dará una función a usar en el videojuego
		Posibilidad de probar varios diseños
6	Hablar sobre los primeros pasos del proyecto	Empezar a trabajar en memoria
7	**Próxima reunión**	20/07/2020

Resumen de acuerdos

Número	**Acuerdo**	**Plazo**	**Responsable**
001	Empezar a trabajar en memoria	Indefinido	Jorge Lacort Navarro
002	El proyecto dará una función a usar en el videojuego	Indefinido	Jorge Lacort Navarro
003	Posibilidad de probar varios diseños	Indefinido	Jorge Lacort Navarro
004	Pensar en las herramientas necesarias para el proyecto	20/07/2020	Antonio Iglesias Soria

REUNIÓN:	**HERRAMIENTAS BÁSICAS**		

Fecha:	20/02/2019		
Hora comienzo:	23:00	**Hora finalización:**	01:00
Lugar:	Universidad San Jorge		
Elabora acta:	Jorge Lacort Navarro		
Convocados:	Antonio Iglesias Soria		

Orden del día / Acta

No.	**Asunto**	**Acuerdo**
1	**Disculpas por ausencia**	
2	**Aprobar última acta**	Lectura y aprobación
3	**Asuntos pendientes última acta**	
4	Empezar a trabajar en memoria	Habrá que hacer ciertos cambios a la planificación
5	Pensar en las herramientas necesarias para el proyecto	Se hará directamente en el proyecto principal
9	**Otros asuntos**	
	Comentar próximos pasos	Continuar trabajando en memoria
		Empezar implementación base
10	**Próxima reunión**	Indefinido

Resumen de acuerdos

Número	**Acuerdo**	**Plazo**	**Responsable**
001	Continuar trabajando en memoria	Indefinido	Jorge Lacort Navarro
002	Empezar implementación base	Indefinido	Jorge Lacort Navarro
003			
004			
005			
006			

REUNIÓN: IMPLEMENTACIÓN BASE	

Fecha:	03/08/2019		
Hora comienzo:	23:00	**Hora finalización:**	01:00
Lugar:	Universidad San Jorge		
Elabora acta:	Jorge Lacort Navarro		
Convocados:	Antonio Iglesias Soria		

Orden del día / Acta

No.	**Asunto**	**Acuerdo**
1	**Disculpas por ausencia**	
2	**Aprobar última acta**	Lectura y aprobación
3	**Asuntos pendientes última acta**	
4	Continuar trabajando en memoria	Acabado estado del arte, objetivos y metodología
5	Empezar implementación base	Acabado, múltiples diseños para el problema han sido discutidos
9	**Otros asuntos**	
	Comentar próximos pasos	Implementar soluciones diseñadas
		Documentar los diseños
10	**Próxima reunión**	Indefinido

Resumen de acuerdos

Número	**Acuerdo**	**Plazo**	**Responsable**
001	Implementar soluciones diseñadas	Indefinido	Jorge Lacort Navarro
002	Documentar los diseños	Indefinido	Jorge Lacort Navarro
003			
004			
005			
006			

REUNIÓN: PROBLEMAS CON EL DISEÑO	

Fecha:	10/08/2019		
Hora comienzo:	19:00	**Hora finalización:**	21:00
Lugar:	Universidad San Jorge		
Elabora acta:	Jorge Lacort Navarro		
Convocados:	Antonio Iglesias Soria		

Orden del día / Acta

No.	**Asunto**	**Acuerdo**
1	**Disculpas por ausencia**	
2	**Aprobar última acta**	Lectura y aprobación
3	**Asuntos pendientes última acta**	
4	Implementar soluciones diseñadas	Discusión sobre múltiples problemas encontrados con el primer diseño, se ha decidido descartar el resto
5	Documentar los diseños	Diseños documentados en memoria en parte de implementación, todo correcto
9	**Otros asuntos**	
	Comentar próximos pasos	Continuar diseño implementado y arreglar problemas
		Documentar solución a los problemas
10	**Próxima reunión**	Indefinido

Resumen de acuerdos

Número	**Acuerdo**	**Plazo**	**Responsable**
001	Continuar diseño implementado y arreglar problemas	Indefinido	Jorge Lacort Navarro
002	Documentar solución a los problemas	Indefinido	Jorge Lacort Navarro
003			
004			
005			
006			

REUNIÓN: CAMBIOS DE DISEÑO Y CASOS EXCEPCIONALES			
Fecha:	17/08/2019		
Hora comienzo:	18:00	**Hora finalización:**	20:00
Lugar:	Universidad San Jorge		
Elabora acta:	Jorge Lacort Navarro		
Convocados:	Antonio Iglesias Soria		

Orden del día / Acta

No.	**Asunto**	**Acuerdo**
1	**Disculpas por ausencia**	
2	**Aprobar última acta**	Lectura y aprobación
3	**Asuntos pendientes última acta**	
4	Continuar diseño implementado y arreglar problemas	Discusión sobre nuevos problemas, posiblemente requerirá más tiempo la implementación
5	Documentar solución a los problemas	Aún pendiente
9	**Otros asuntos**	
	Comentar próximos pasos	Implementar cambios mencionados al diseño
		Documentar solución a los problemas
10	**Próxima reunión**	Indefinido

Resumen de acuerdos

Número	**Acuerdo**	**Plazo**	**Responsable**
001	Implementar cambios mencionados al diseño	Indefinido	Jorge Lacort Navarro
002	Documentar solución a los problemas	Indefinido	Jorge Lacort Navarro
003			
004			
005			
006			

REUNIÓN: CASO DE ENCONTRAR LÍNEAS CERRADAS	

Fecha:	31/08/2019		
Hora comienzo:	22:00	**Hora finalización:**	24:00
Lugar:	Universidad San Jorge		
Elabora acta:	Jorge Lacort Navarro		
Convocados:	Antonio Iglesias Soria		

Orden del día / Acta

No.	**Asunto**	**Acuerdo**
1	**Disculpas por ausencia**	
2	**Aprobar última acta**	Lectura y aprobación
3	**Asuntos pendientes última acta**	
4	Implementar cambios mencionados al diseño	Rechazado, al final no lo de las líneas cerradas no es posible dentro de las condiciones dadas
5	Documentar solución a los problemas	Aun por realizar
9	**Otros asuntos**	
	Comentar próximos pasos	Arreglar casos extremos y depurar la funcionalidad
		Acabar la memoria
10	**Próxima reunión**	Indefinido

Resumen de acuerdos

Número	**Acuerdo**	**Plazo**	**Responsable**
001	Arreglar casos extremos y depurar la funcionalidad	Indefinido	Jorge Lacort Navarro
002	Acabar la memoria	Indefinido	Jorge Lacort Navarro
003			
004			
005			
006			

REUNIÓN: ARREGLOS FINALES	

Fecha:	09/09/2019		
Hora comienzo:	22:00	**Hora finalización:**	24:00
Lugar:	Universidad San Jorge		
Elabora acta:	Jorge Lacort Navarro		
Convocados:	Antonio Iglesias Soria		

Orden del día / Acta

No.	**Asunto**	**Acuerdo**
1	**Disculpas por ausencia**	
2	**Aprobar última acta**	Lectura y aprobación
3	**Asuntos pendientes última acta**	
4	Arreglar casos extremos y depurar la funcionalidad	Acabado, se han conseguido arreglar todos los problemas que había hasta el momento
5	Acabar la memoria	Aún por realizar
9	**Otros asuntos**	
	Comentar próximos pasos	Acabar la memoria
		Acabar estructura de memoria
10	**Próxima reunión**	10/09/2020

Resumen de acuerdos

Número	**Acuerdo**	**Plazo**	**Responsable**
001	Acabar la memoria	11/09/2020	Jorge Lacort Navarro
002	Acabar estructura de memoria	10/09/2020	Jorge Lacort Navarro
003			
004			
005			
006			

REUNIÓN:	**ARREGLOS FINALES**

Fecha:	10/09/2019		
Hora comienzo:	22:00	**Hora finalización:**	24:00
Lugar:	Universidad San Jorge		
Elabora acta:	Jorge Lacort Navarro		
Convocados:	Antonio Iglesias Soria		

Orden del día / Acta

No.	**Asunto**	**Acuerdo**
1	**Disculpas por ausencia**	
2	**Aprobar última acta**	Lectura y aprobación
3	**Asuntos pendientes última acta**	
4	Acabar estructura de memoria	Acabado, ciertos ajustes mencionados, pero visto bueno para acabar y entregar
5	Acabar la memoria	Aún por realizar
9	**Otros asuntos**	
	Comentar próximos pasos	Acabar la memoria y entregar
10	**Próxima reunión**	Indefinido

Resumen de acuerdos

Número	**Acuerdo**	**Plazo**	**Responsable**
001	Acabar la memoria y entregar	11/09/2020	Jorge Lacort Navarro
002			
003			
004			
005			
006			

Annex C

DAILY HOURS

DATE	HOURS
20/07/2020	8
21/07/2020	8
22/07/2020	6
28/07/2020	8
29/07/2020	6
30/07/2020	8
31/07/2020	8
03/08/2020	8
04/08/2020	8
05/08/2020	8
06/08/2020	8
07/08/2020	8
08/08/2020	4
10/08/2020	8
11/08/2020	8
12/08/2020	8
13/08/2020	8
14/08/2020	8
17/08/2020	8
18/08/2020	8
19/08/2020	8
20/08/2020	8
21/08/2020	8
24/08/2020	8
25/08/2020	8
26/08/2020	8
27/08/2020	8
28/08/2020	8
02/09/2020	8
03/09/2020	8
04/09/2020	8
05/09/2020	8
06/09/2020	8
07/09/2020	12
08/09/2020	8
09/09/2020	8
10/09/2020	8
11/09/2020	8