Lucy Lesire

Real time changes in sketch based RTS formations

Graduation work 2022-2023

Digital Arts and Entertainment

Howest.be

CONTENTS

ABSTRACT

This paper presents real time changes in a custom formation system, to improve formations in existing real time strategy games. The project allows the user to use simple drawings to create real time strategy game formations in real time. At the same time, the project also allows for the user to remove units in existing formations, causing the formation to reform the original formation, but with less units. The project recognized user input drawings by use of a gesture recognition algorithm, allowing the user to create formation templates and let them be recognized later. The formations are also improved by the use of a grid, allowing for filling of formation and the removement of minor unwanted flaws in the drawing. The project ensures that the scaling of the formation is always as optimal as possible to the number of units in the formation. In this paper you will be able to see that the scaling will always account for all units, with a marge of 2 units. The real-time changes in the project allow for direct implementation in to existing real time strategy games, to enhance formation control.

INTRODUCTION

In Real Time strategy (RTS) games, formations are often represented by simple squares or rectangles. However, RTS games are very tactical and could benefit from more control in formations, like the ability to change your formation in a way that it would avoid large attacks (e.g., cannons). In the current RTS games, the ability to create custom formations is either not present or very tedious like moving each individual unit [1]. In this paper, a way to create custom formations by drawing on the screen is approached. It makes use of gesture recognition, grids, and scaling. This paper also goes into detail on changes in custom formations. In RTS games, units can die, and if these units die whilst in a formation, the formation ends up with a hole, no reconstruction of the formation is done. This is why the driving question behind this paper is: **Can custom sketch based RTS troop formations react and adjust to changes in real time?**

RELATED WORK

1. CROWD SIMULATION

Crowd simulation is the process of simulating the movement and/or the behavior of a large number of entities or characters [2]. Recently, crowd simulation has had a lot of advances, mainly in the following categories: crowd evacuation, pedestrian crowds, traffic simulation, and swarm simulation [3]. All of which are different forms of crowd simulation. Although there have been a lot of recent advances in crowd simulation, it is still a relatively young research area. This comes due to the complexity and variety of real-world crowd behaviors, this causes most crowd simulation systems to be application specific.

One of the earliest crowd simulations is the rule-based model proposed by Reynolds [4]. Reynolds used a “boid” system to simulate a herd of birds and a school of fishes. Reynolds system was influenced by the particle system [5]. The flocking of birds is an interesting concept, it uses a combination of rules or behaviors. The most prominent are Coherence, Alignment, and separation [6], these behaviors influence individual entities in the crowd, and thus also influence the entire crowd. Reynolds later expanded this method using spatial partitioning and spatial hashing for large crowds on the PS3® [7].

Some of the more recent methods of crowd simulation make use of the following concepts: Cellular Automaton and Collision avoidance.

1.1 INDIVIDUAL PERSONAS

A crowd is made up of a (large) collection of entities or objects. Entities in a crowd can be influenced by a handful of personal factors: physiological, psychological, social and primal. Depending on these factors, each entity will behave differently in a crowd. However, these entities are not only influenced by all these personal factors, but they are also influenced by global factors, which can usually be seen as the driving factor of a crowd.

A personal factor can include an entity's personality, if an entity is aggressive in personality, they might assert more dominance in a crowd, this can especially be seen in panic simulations. In a panic simulation, an aggressive entity will have no regard for the entities around them, possibly knocking them over or pushing them out of the way [8].

A global factor can be the reason why the crowd exists in the first place. In a panic simulation, the global factor will be getting out of the panic location.

In games, these individual factors are often stereotypical and will most of the time exists of one characteristic: "angry, shy, stupid, …" . Whereas in real life, these individual factors are often in large numbers, and can change from day to day. They can be categorized into 2 classes, external and internal factors.

2. RTS FORMATIONS

In real-time strategy (RTS) games, such as Age of Empires, StarCraft, Total War: Warhammer, players need to control their military units to attack opponents, and win the game. Often these formations are controlled by simple mouse drags and clicks. However, these formations are often simple shapes like squares or rectangles.

When approaching the ability to have more custom formations, we fall into the category of Group Formation, a subcategory of Crowd Simulation. One of the advances that have recently been made in the field of custom formations, is the ability to deform a group formation by pinning or dragging units in this formation, while maintaining the formation boundaries [9]. There has also been an approach to generate custom stylized formations by using formation boundary sketching [10].

A very recent approach makes use of gesture recognition and formation templates to convert sketch-based drawings into formations in real time [11]. This paper has inspired the use of gesture recognition in this project. However, without the use of formation templates.

3. GESTURE RECOGNITION

Gesture recognition is focused on recognizing hand gestures through mathematical algorithms. These gestures come from the movement of body parts on the human body, most of the time coming hands, face or arms. By using gesture recognition, the user can control or interact with digital devices. Gesture recognition can currently be used with certain VR devices that allow for hand recognition to control their devices.

Gesture recognition can be done through 2 main approaches, a parametric approach, and a template approach. The parametric approach extracts different features from the input gestures and uses classification tasks [12]. The most common template approach compares the input gesture with a defined set of gesture templates [12].

Some gesture recognizers that use a template approach are the "$1 recognizer" and the "$ family" [13]. These recognizers make use of a four step algorithm for gesture recognition. When using a $ recognizer, the user only has to define one gesture as a template, after this they can start using this template to recognize similar gestures.

However, before the template can be used, the $ family uses various preprocessing steps, scaling, sub-sampling, and alignment of gesture directions. These preprocessing steps provide better performance and accuracy when using the recognizer[12]. The $ family recognizers are also very well suited in a scenario where the user's gestures are relatively simple and don't have a lot of templates.

One recognizer in the $ family that stood out in this project is the "$P Point-Cloud recognizer", this recognizer makes uses of point-clouds to store their gestures and templates [14]. A point cloud is a dataset that represents a point in space. The $P Point-Cloud recognizer ($P), makes use of the Greedy-5 algorithm to recognize gestures. Greedy-5 uses Euclidian distances with weights to match points in 2 sets of point clouds [14]. $P Allows for the recognition of both Uni- and multistroke gestures. It also removes one of the problems of a previous member in the $ family, the $1 recognizer, and the directionality problem. The directionality problem was caused by the recognizer misrecognizing certain gestures if they are drawn in an opposite direction than the template. $P does not have this problem, as it uses Point-Clouds without direction to store their points. The $P recognizer is able to recognize multistore gestures more performant that the $N recognizer [14] [15].

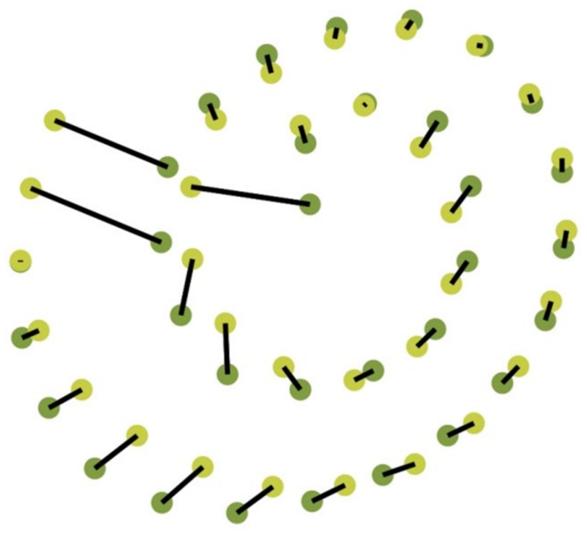

Figure 1: The $P Point-Cloud Recognizer using Greedy-5 to match two sets of point clouds, gesture and template [14] **(2012).**

CASE STUDY

1. INTRODUCTION

Formations in Real Time Strategy (RTS) games are often shaped simple, squares, rectangles, circles. If the player of an RTS game would like to have more complex formation shapes, they would have to manually start moving each unit to fit how they like, a tedious and annoying work. However, there are other ways to approach custom formations, in this system we explore: (1) Formation Templates using gesture recognition; (2) Formation improvement using a grid; (3) Formation scaling; (4) Real time changes.

2. FORMATION TEMPLATES USING GESTURE RECONITION

Formation templates are inherently a simple concept, the user defines what they want their formation to look like, this can be done by drawing the formation on the screen. The user can then choose to save this formation and for it be added to a list of templates. If the user then wants to call up this formation template, they simply have to draw a sketch of their formation and the program will use gesture recognition to get the correct template.

2.1 GESTURE RECOGNITION USING $P

Gesture recognition is used almost daily in the current world, a simple gesture can be drawn on your smartphone and your flashlight will turn on. This project also makes us of the gesture recognition world, because we would want the user to design their own recognition templates, it needs to be an algorithm that can handle having small sample sizes.
The $P Gesture recognizer [14] is a point-cloud based recognizer that is designed for small and rapid gesture recognition. By using point-clouds $P is able to recognize both unistroke and multistrokes gestures, a perfect fit for our project, as we want to give the user the freedom of drawing what they want.

Some advantages of using the $P Point-Cloud recognizer over some of the other recognizer in the $ family include: (1) No directionality problem, the drawing can be drawn in any direction, $P stores the gesture as a point-cloud as can be seen in figure 2; (2) Fast computing for gestures with multiple strokes.

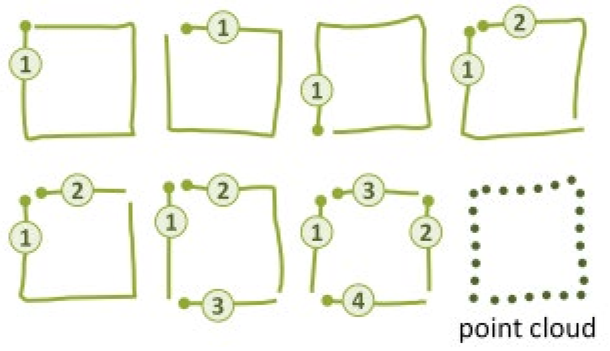

Figure 2: "Even a simple square can be drawn using 1,2,3, or 4 strokes which can vary in order and direction (with a total of 442 possible cases). However, all the articulation details are ignored when looking at the square as a time-free cloud of points." [14] **(2012).**

2.1.1 STROKED PATH RESAMPLING

For the recognizer to be most efficient, it approaches resampling of gestures. When comparing two gestures both gestures could have different amounts of points, this could be the result of the speed of the drawing, the size of the drawing and the device used to draw. Therefore it is important that the recognizer resamples each gesture to have the same amount of points before matching the gesture with the templates. This sampling size can determine how detailed a gesture is and thus how good the recognizer works but it can also impact the speed of the gesture recognition algorithm. As the sampling size increases, the recognizer will run slower, but the gestures matching becomes more accurate. In figure 3, an example of the different sample sizes can be seen for one gesture.

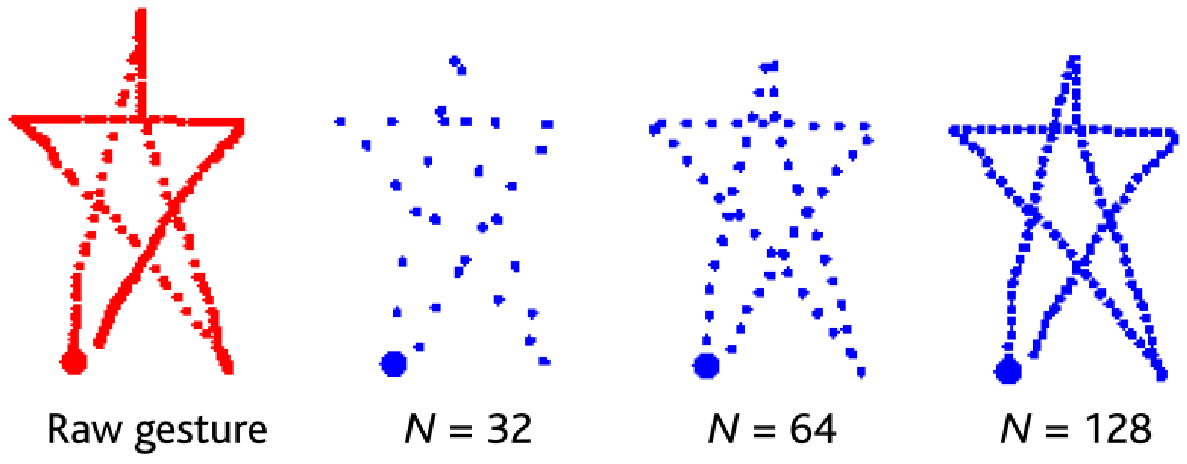

Figure 3: A star gesture resampled to N=32, 64, and 128 points [13] **(2007).**

2.2 TEMPLATES TO FORMATIONS

By using the $P Gesture recognizer, our drawn gestures will be stored as templates. When the user wishes to form a formation with said template, an algorithm will have to define where each unit will have to be.

Since our gestures are stored as a series of cloud points, if our number of units is the same as the sample size of our gestures, we can simply assign each unit to a point.
However, if our number of units is smaller than our sample size, we will have to loop over the number of points in our gesture according to the number of units. This can be done by using the following equation:

- Loop iterator = sample size / number of units

With this iterator we can loop over our point so that our units will be evenly spaced on our drawing.
If the number of units is larger than the sampling size, there could be 2 solutions:

1. Limiting the number of units to our sample size (this would give us formations with a maximum of sample size units)
2. Increasing the sample size of our gesture.

3. IMPROVED FORMATIONS USING A GRID

By approaching our troop formations with just the points of the $P Point-cloud recognizer, the result will be the outline of the gesture being used as a formation. However, to have a more complete formation, the inside also must be accounted for. A way to approach this is by using a background grid.

3.1 GESTURE OUTLINE ON THE GRID

The $P Point-cloud recognizer will give us a series of points, how large this series is, is dependent on the sample size. If a grid is outlaid behind our gesture, we can map the points of our gesture to the grid as can be seen in figure 4.

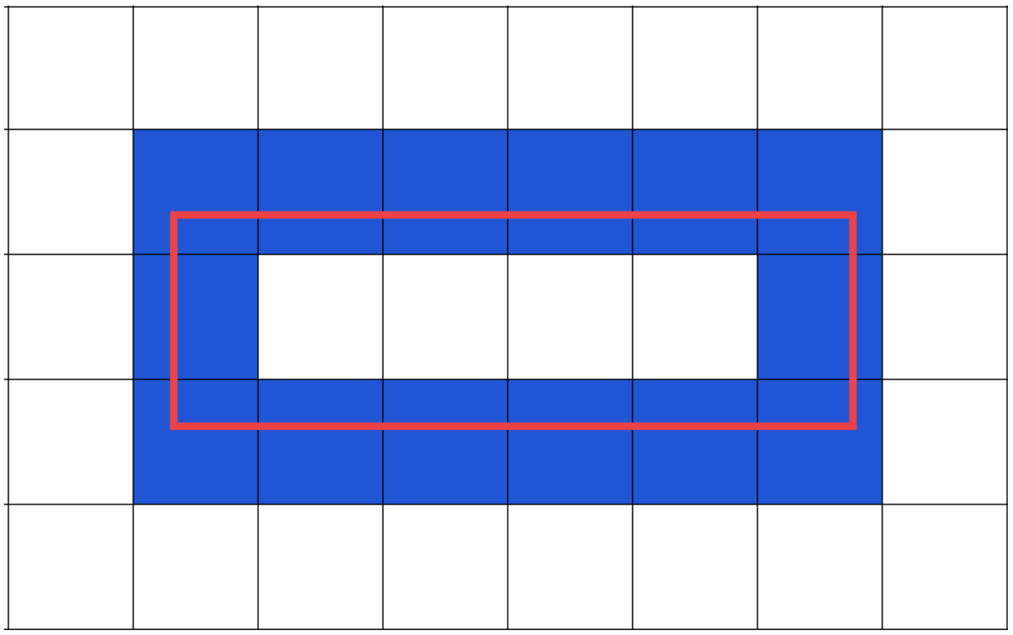

Figure 4: An example of mapping the points of a rectangular gesture to a grid.

By mapping our gesture points to this grid, we can solve 2 issues that our initial proposed system had:

1. Removing unnecessary details in the gesture that were created due to shaking when drawing the gesture
2. Determine the space inside gestures.

3.2 THE FLOOD FILL ALGORITHM

To know the inside of a drawn shape on a grid, a flood fill algorithm can be used. The flood fill algorithm is a simple flooding algorithm that will fill up an area. It is used in a lot of common practices like the “bucket” fill tool in Photoshop. By using the flood fill algorithm (Now used simply as flood fill), we can determine if our shape is open or closed. The flood fill algorithm will start from one tile on a grid and check its neighbors, if the neighbors are empty, then the flood fill algorithm will “activate” the neighboring tile and check their neighbors. This will keep going until there are no more tiles to check. To steer the flood fill algorithm, some tiles on the grid are known as “blockers”. These “blockers” will block the continuation of the flood fill algorithm, not allowing it to continue this tile.

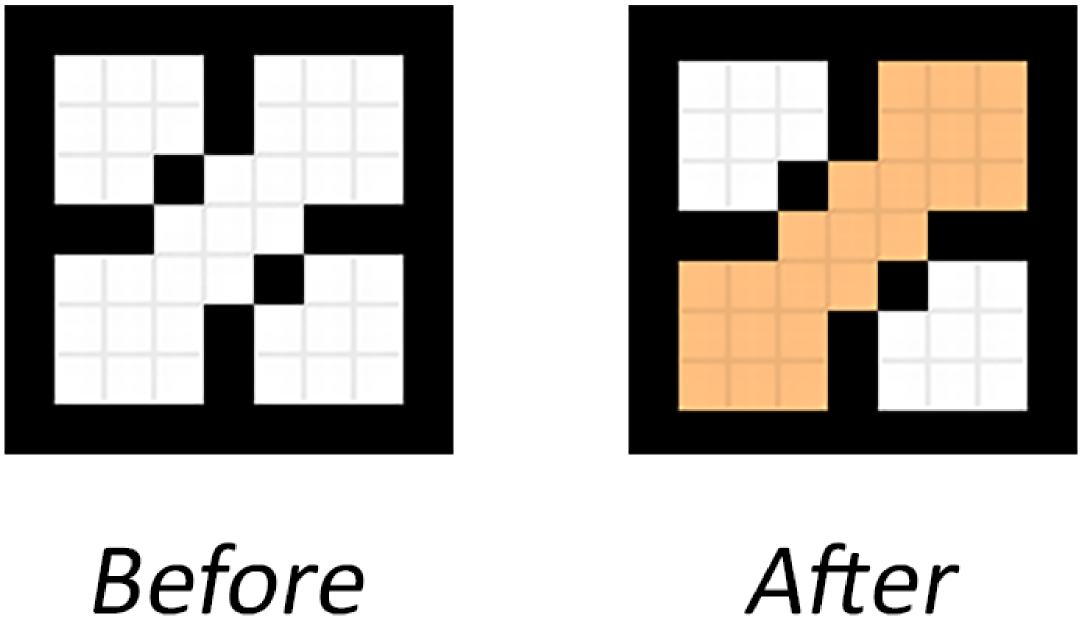

Figure 5: an example of the flood fill algorithm [16].

In figure 5 we can see an example of the flood fill algorithm on a grid [16]. The black tiles are "blockers", the white squares are inactivated tiles, and the orange tiles are activated. The flood fill algorithm in this example started from the center tile.

In our context, if we define the "blockers" as the tiles that the points have been mapped to and start our flood fill from the top right corner. The result will be that a closed shape will have their tiles on the inside untouched. These untouched tiles can then be taken and used to fill in the inside of our formation. However, if there are no untouched tiles then the shape is not closed and thus has no inside.

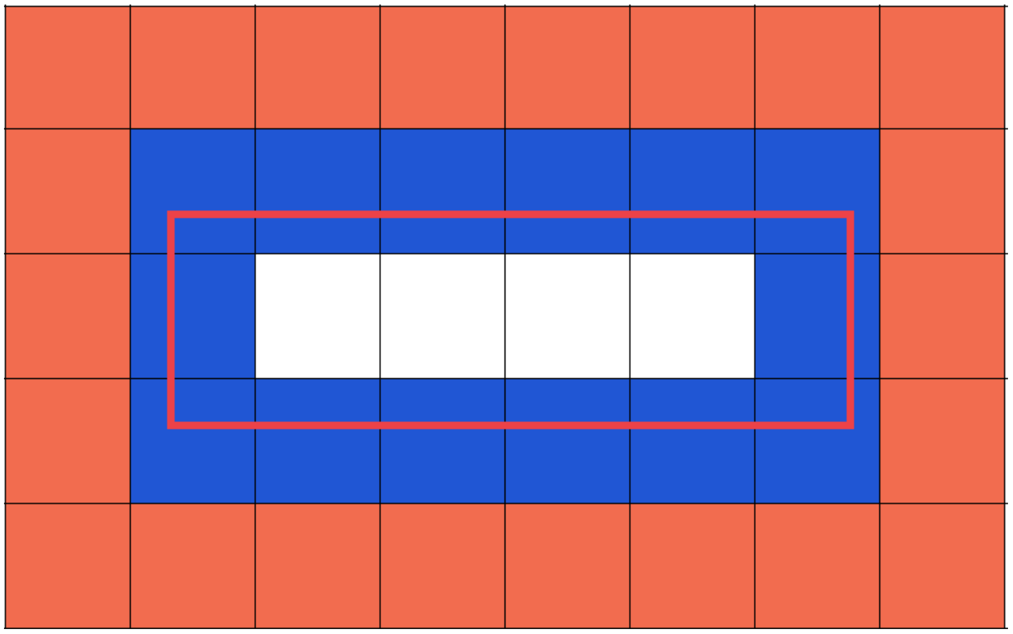

Figure 6: The flood fill algorithm applied to the closed shape in figure Y, resulting in untouched tiles.

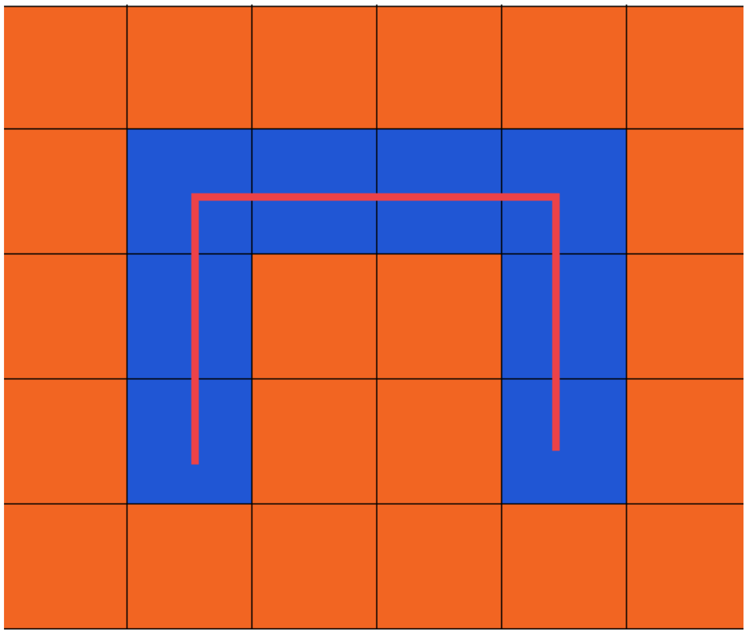

Figure 7: The flood fill algorithm applied to an open shape, resulting in no untouched tiles.

When mapping our gesture to the grid, we must consider that the size of the grid can determine the complexity of the mapped shape and the speed of the flood fill algorithm. The following rule applies: As the tile size decreases, the gesture becomes more detailed, and the algorithm becomes slower.

4. FORMATION SCALING

It will often happen that our unit amount will not fit the sample size of the gesture or the number of tiles on the grid. When too many units are present, we will see an “over fill”, too many units will be present to fit the formation. When too little units are present, we will see an “under fill”, too little units will be present to fit the formation properly.

4.1 OVERFILL

When an overfill happens, the sample size of the gesture will have to be increased to account for more units. The formula for this can be approached very easily as a 1:1 ratio. Sample size = unit amount.

On the other hand, when the grid approach is used and an overfill happens, we will have to scale up the gesture so that enough tiles are present for all the units. The formula for this will have to be further experimented as it is not just a 1:1 ratio.

4.2 UNDERFILL

When an underfill happens, the sample size of the gesture does not have to be adjusted. Depending on the sample size of our gesture, nothing may have to change anything at all. E.g., with a sample size of 64 and a unit amount of 32, the loop iterator rule must be followed to make sure that the troops are equally distant from one another. When we have a unit amount that is much lower than our sample size (e.g., sample size of 64 and unit amount of 10), we will have to scale down the gesture so that our units can still represent the shape of the gesture.

With the grid approach, when an underfill is present, we will also have to scale down the gesture, this will make sure that the units can still represent the gesture as good as possible.

4.2.1 TROOP CONVERSION

One problem does keep being apparent in the underfill when using the grid approach, the inside. The plan is to use a different type of unit for the inside, then for the outside. What might happen is that there are so many outside units compared to inside units, that it will appear as if the formation is barely filled.
This is why I looked at troop conversion, with troop conversion, outside or inside units will be able to “change role” to fit as an inside or outside unit. This way when scaling, we will have to make sure that there is always enough room for all units and that we always get a complete formation.

5. REAL TIME CHANGES

The final thing present in this project are real time changes. In RTS games, it can often happen that units die in combat. When a unit were to die, the formation should not be running around with a hole in it. Instead the formation should adapt to their lost member and reform the formation with less units.

5.1 DYING

When a unit dies, the formation will have to notice that a unit is no longer in the formation, then they should reform the formation depending on the gesture that the user had input, without any intervention by the user and in real time. When this happens, the formation will most likely have to scale down to prevent under filling.

EXPERIMENTS & RESULTS

1. GESTURE RECOGNITION

The initial implementation of the $P Point-Cloud recognizer in Unity can be found on the Unity Asset Store [17]. This implementation came with the following 3 things:

1. The gesture recognition algorithm.
2. A draw area for the user to draw.
3. An “Add as” button for the user to add their drawing as a template.

This implementation was very basic and was able to recognize shapes. The result of the recognition would be a text-based output of the recognized template as text, as well as a “comparison ratio”, which stated how accurate the recognition was. The “add as” button would allow the user to add their own templates, something very useful for this project.

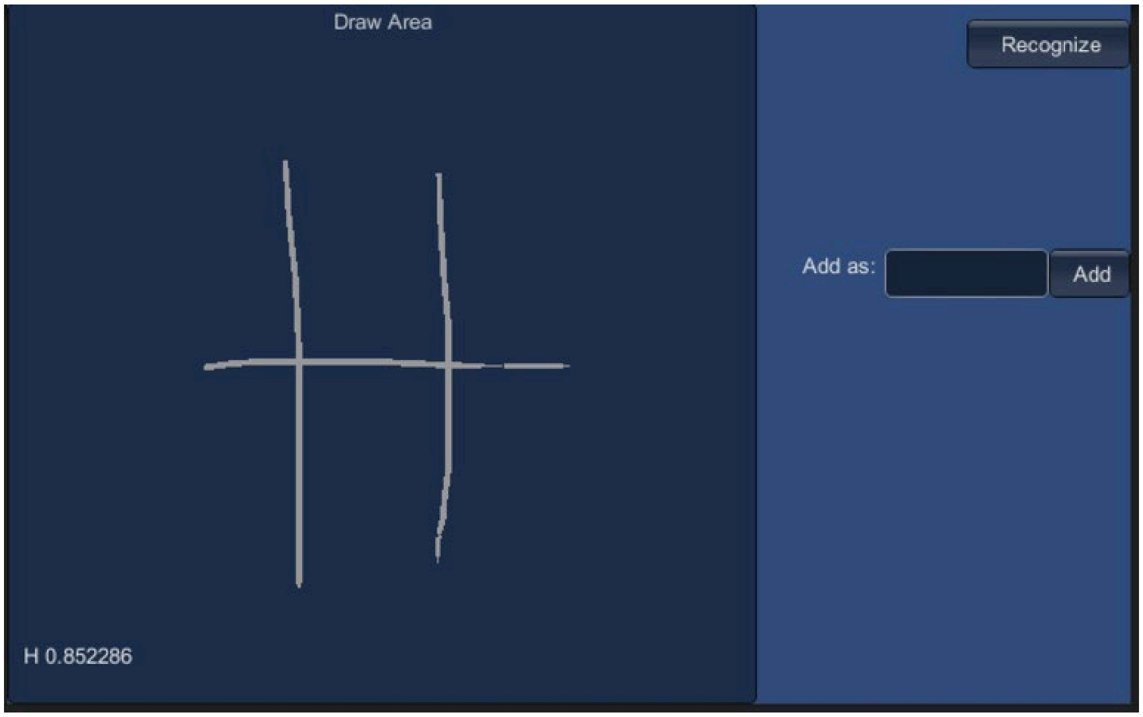

Figure 8: PDollar Point-Cloud Gesture recognizer (2015) [17].

This implementation was however not yet enough to start creating formations. The first feature that I wanted to add was a visual print of the gesture that had been recognized, upgrading the text-based output to a visual output. This was done by using the “onscreen gesture” draw of the Unity engine, following the points that had been saved in the XML file of the recognized template.

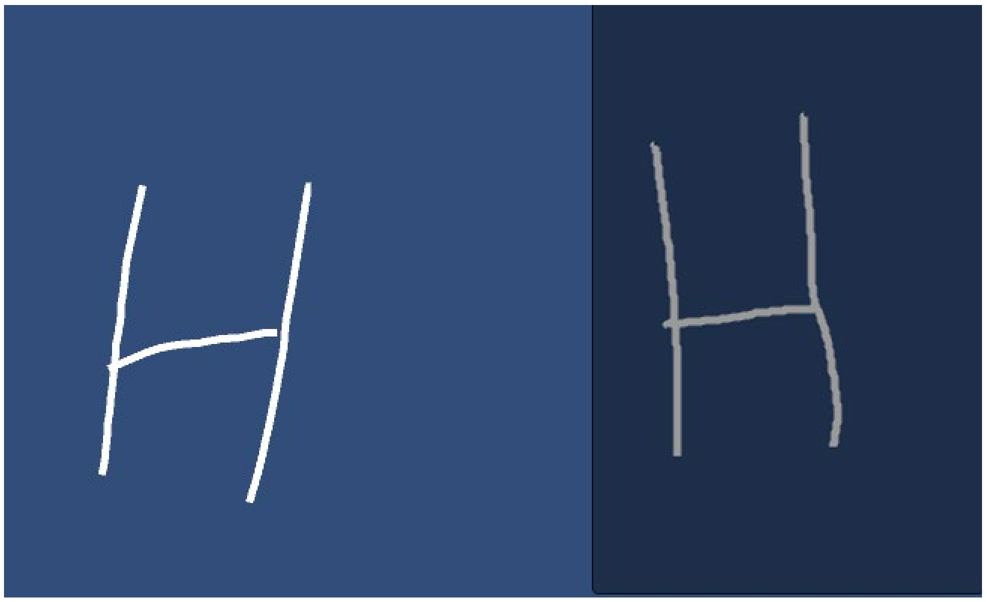

Figure 9: Sketch shape drawn on the screen.

The final feature I wanted to add to this was the ability to add “child templates” to already existing templates. This would allow for one template to be recognized with multiple gestures. E.g., if the user decides they want to have a template of an ‘H’ and want both the ‘H’ and ‘h’ to be recognized as the ‘H’, they can add the small ‘h’ gesture as a “child template” of the big ‘H’, this example can be seen in figure 10.

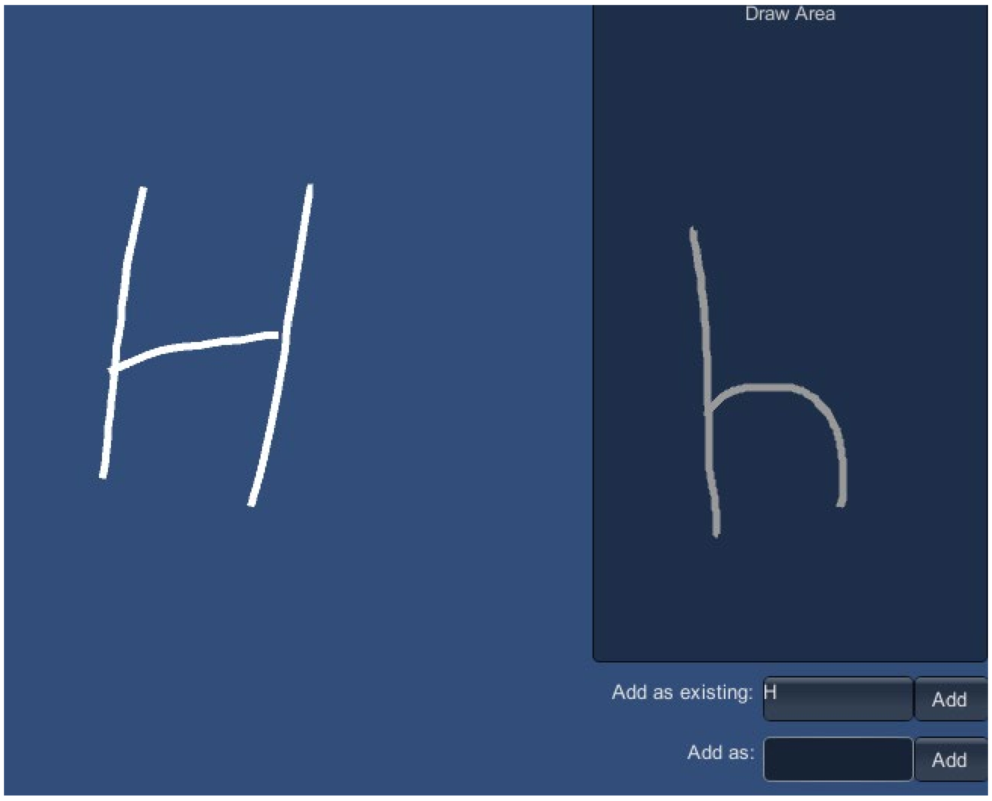

Figure 10: Using a small letter to recognize a block letter (child-shapes).

2. FORMATIONS WITH GESTURE RECOGNITION

For all the further experiments a sample size of 64 is used to test the recognition algorithm.

The first tests for creating the custom formations on screen, was to use the 64 points in the gesture and to assign one unit to each of these points. However, quickly one of the earlier managed problems arises, when we don't have equal units as our sample size (64), the formation does not look like the template at all. To have a more accurate formation to the template, an exact number of units as the sample size is needed. To fix this problem, a "Loop Iterator" was created, this loop iterator is equal to the sample size divided by the unit amount. With this iterator, the units would be looped over the points, creating an equal distance between the points to get a clear formation.

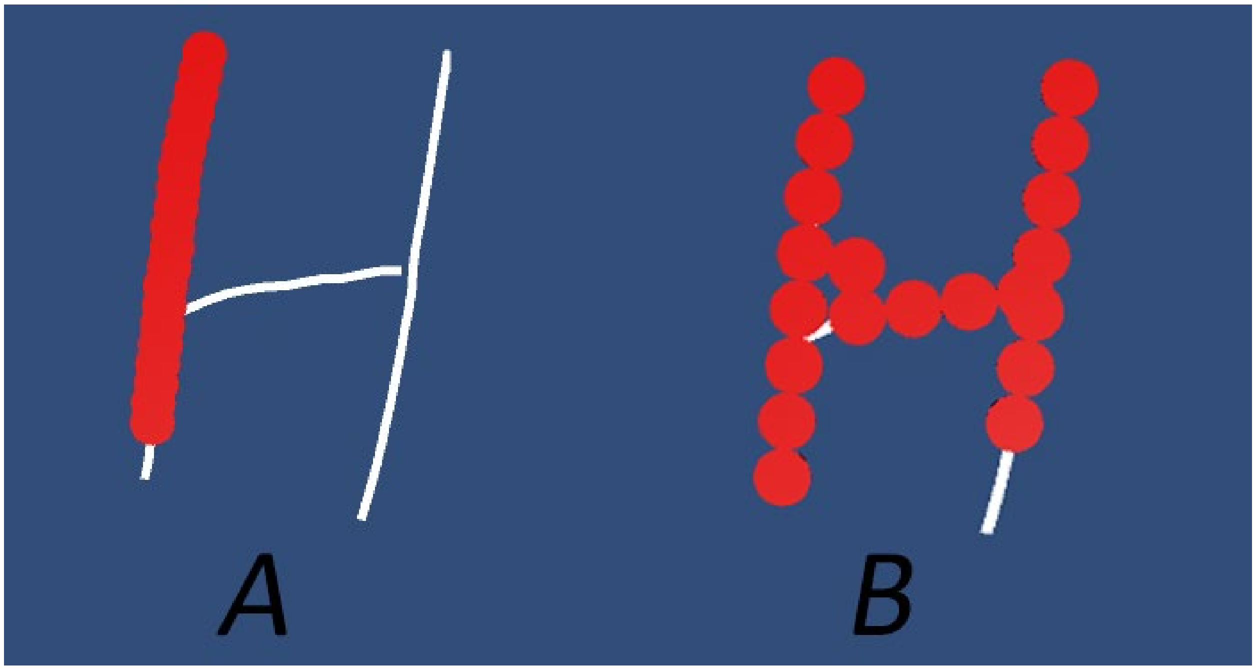

Figure 11: Simple overlay vs loop iterator method.

In figure 11 you can see that option A gives the user a formation that barely looks like the template ('H'). Option B used the "loop iterator" and resembles the template ('H') a lot closer.

When trying out this method, a problem quickly becomes apparent, the formation relies entirely on the template that the user had drawn with the hand. Since humans are often not the most precise and correct when using a mouse, a template can have a lot of unwanted details and crooked lines. An example of this impreciseness can be seen in figure 12, where the template is crooked.

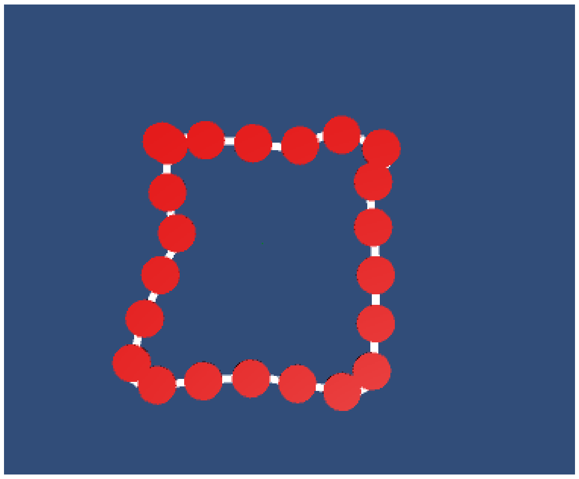

Figure 12: Deformed Square.

Another problem that had been encountered is that these formations only have an outline, there are no checks to see if a template is open or closed and there are no troops on the insides of the formations.

To fix both problems, I continued with approaching the grid option.

3. IMPROVED FORMATIONS BY USING A GRID

To use grids, we first must create a grid. In Unity, a grid is simply an array of tiles. The tiles have a scale, which determines the size of the tiles, depending on the size of our tiles we could have more, or less detail in our formations. For the following tests, a grid size of 30x30 has been used, as well as a tile scale of 3.

To turn my templates into “grid templates”, a grid would first be generated. The template would then be overlayed on the grid. When the template is on the grid, a mapping is done to turn the points of the template into points on the grid, as well as removing any double points that are on the same tiles previous points.
This new list of points would then be used to generate the formation. This is once again done by assigning a unit to each point in the list of points and looping over the point according to the number of units (“loop iterator”).

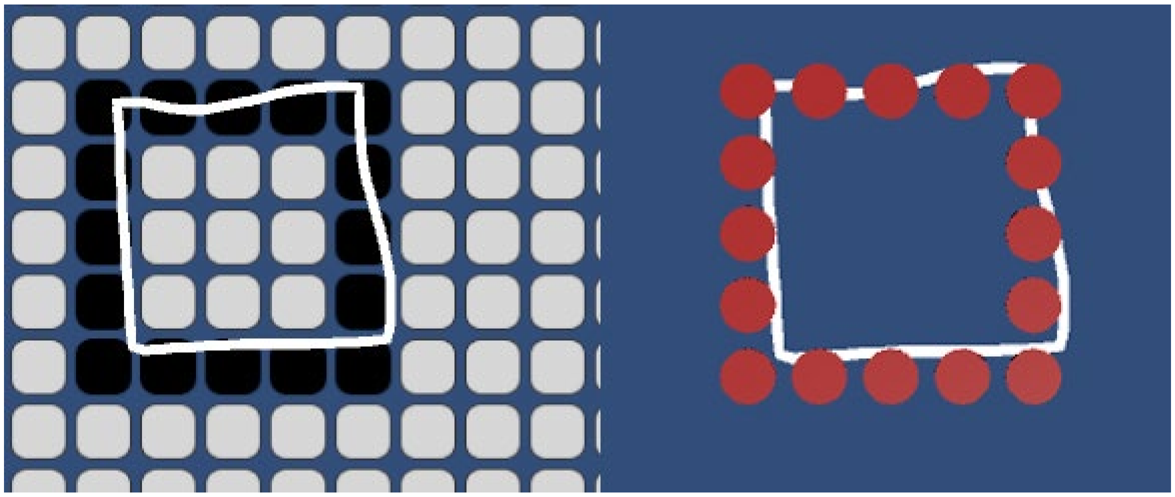

Figure 13: Grid mapping of a template.

In figure 13, the mapping of the template can be seen, each point is mapped to the tile it is on, that tile is then saved in our new list of points (and colored black). Once all points have been mapped, we will create our formation, as can be seen on the right side of figure 13.

One problem is still apparent in our current testing, the “over- and under fill” (4. Scaling). The “under fill” is very recognizable. In figure 14, on option A, we can see that the “under fill” is present, the spacing between our unit is so big that the formation does not look like either the template or “grid template”. The “over fill” can also been seen in figure 14, in option B, there are so many units, that they have started stacking upon one another.

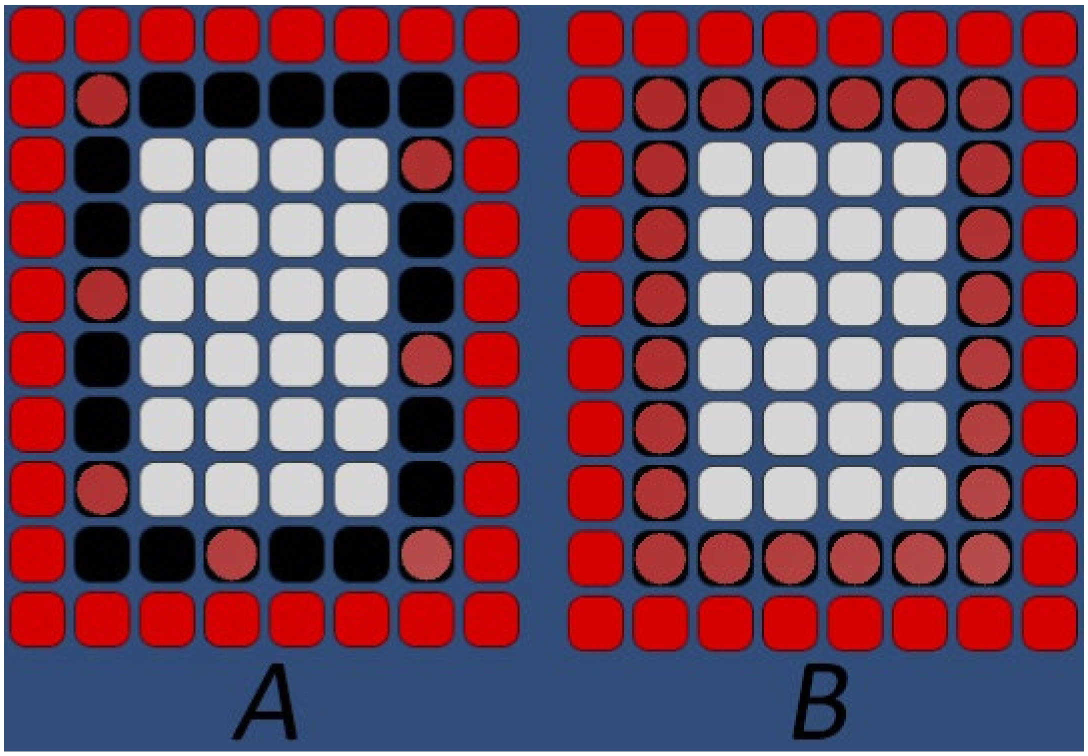

Figure 14: Under and Overfill in the outline.

3.1 THE FLOOD FILL ALGORITHM

After mapping our template to the grid and assigning the mapped tiles as blockers, a flood fill algorithm can now be used on the grid, starting from the top left corner. Since every tile that the flood fill algorithm can pass will be filled by our algorithm, it is possible to check for “untouched” tiles after the flood fill is done. If there are any tiles left over, we have a closed shape, if there are none, it’s an open shape.

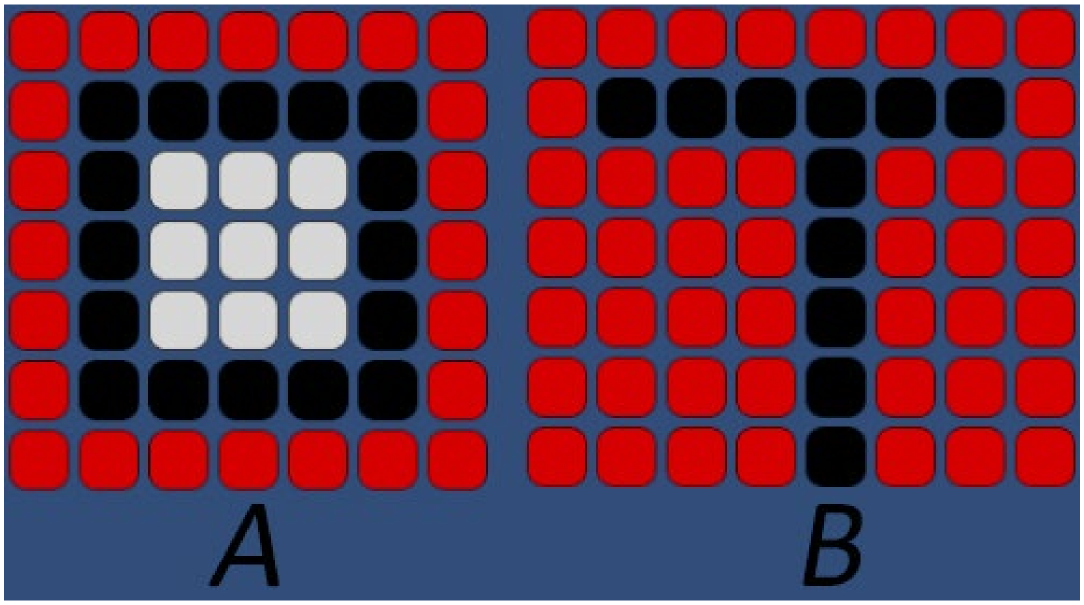

Figure 15: Closed and open shape after flood fill.

In figure 15 we can see 2 examples of the aftermath of the flood fill algorithm. In option A, the tiles inside the square are untouched, meaning that we have a closed template. In option B, all tiles (besides the "blockers") are touched by the algorithm, this means that we have an open template, and no inside.

If our template ended up as a closed template after the flood fil algorithm, we had space to assign units to the inside of the template. The same approach for assign units to the outline is used to assign units to the inside. For the filling of the shapes, I used a blue colored unit that will be referred to as "inside units", the red colored units will be referred to as "outside units".

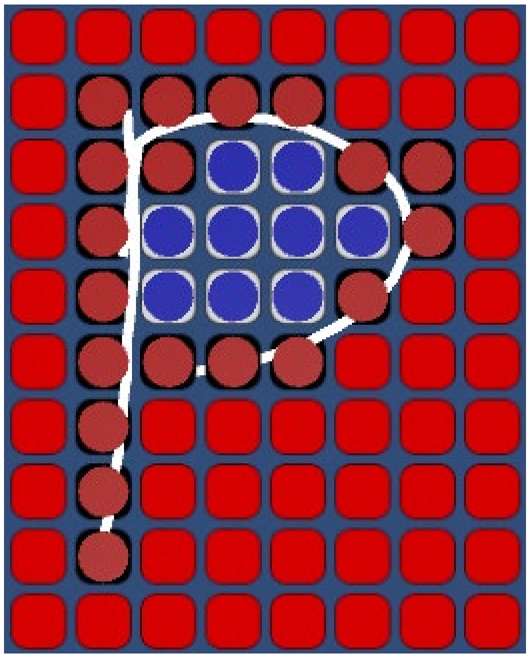

Figure 16: Closed shape with a filled inside.

In figure 16 we can see a filled up closed template (P). However, this example reminds us of an already existing problem, that the inside of the closed template would only be filled if the amount of inside units matches the amount of tiles. With too little units, we would once again get an" under fill" and with too many units, we would get an "over fill". This problem can once again be seen in figure 17.

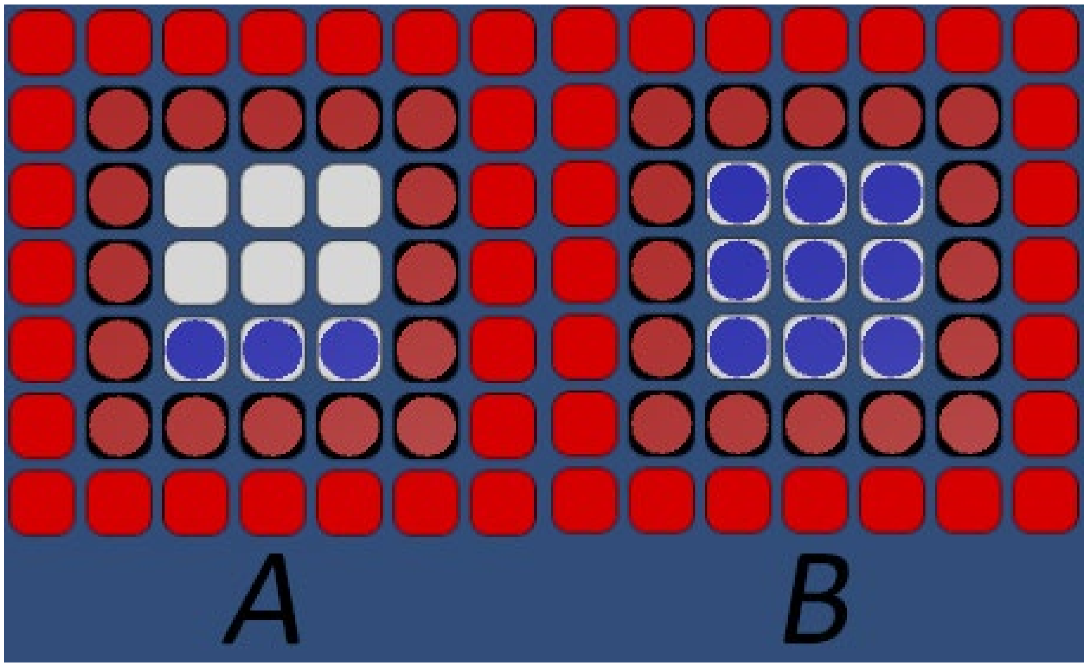

Figure 17: Under- and overfill with inside units.

Since this problem has already occurred multiple times, the next experiments are mainly about scaling, a possible solution to this problem.

4. FORMATION SCALING

The solution to the problem at hand seemed to be simple, scale the formation up when an "overfill" is present and scale the formation down when an "underfill" is present.

For the following tests, I used the square gesture as my scale default.
In table X, the top row represents the number of outside units used to create the formation. The bottom row represents the scale factor per unit to scale the formation, the total scale would then be calculated by multiplying that number with the number of outside units.

Units	4	5	7	10	12	15	20	25
Scale per units	0.28	0.3	0.235	0.17	0.18333…	0.178	0.178	0.178

Table 1: the (outside) unit scale function.

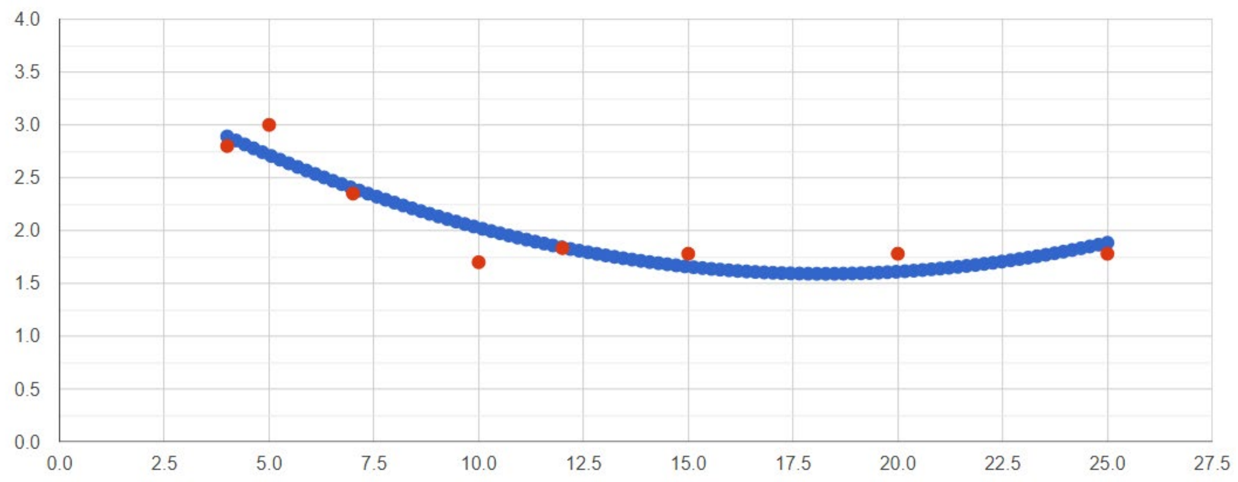

Figure 18: a parabola function build up from the statistics in table 1 (the Y has been scaled by 10 to make the function be more visible).

$$\boxed{f(x) = 0.000671009x^2 - 0.0247235x + 0.379756}$$

Figure 19: The mathematical equation that gave us the parabola in figure 18.

As we can see with table 1 and figure 18, we have a scaling curve that starts high, goes lower, rises again and finally stagnates. With this it was possible to formulate: If the number of units is small, the scale will have to be high. If the number of units is large, the scale will be low, however above roughly 15 units, the scale would almost always be around 0.178 per unit. This function was then used to scale the gesture according to the number of units. This gave me good results, when only accounting for the “outside units”.

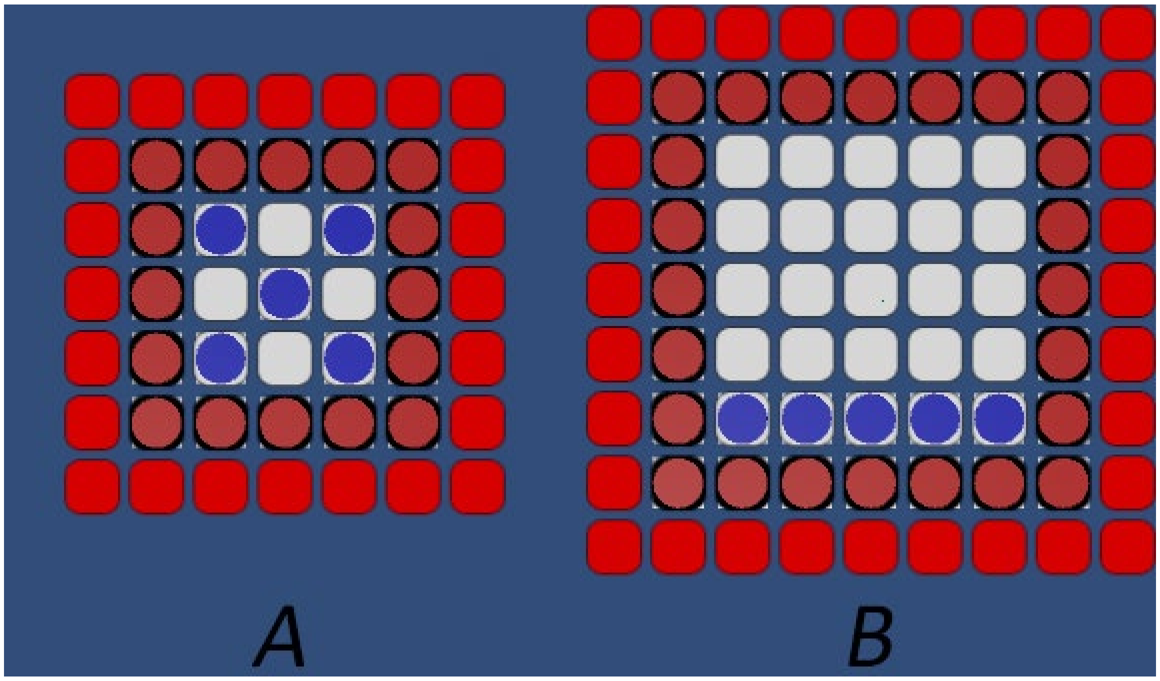

Figure 20: Scaling only applies to outside units.

In figure 20, option A, there are 18 outside units, along with 5 inside units. In option B, there are 24 outside units, along with still 5 inside units. In option B, the formation has been scaled up according to the above formula. However, the inside units have not increased, thus the spacing between them has been increased as there are now more inside tiles after the scaling. This is an undesired effect of the current scaling, because we don't want the inside troops to be organized like that inside of our formation.

This is why I changed the scaling method to account for "Unit Conversion".

4.1 UNIT CONVERSION

Unit conversion means the conversion of one type of unit into another type of unit (e.g., an outside unit becomes an inside unit). By using unit conversion, we can make sure that (almost) all tiles of our template are filled by a unit.

Unit conversion, even if it sounds simple, is not that simple. First, we have to check what the difference is between the total amount of units and total amount of tiles in our formation. To take option B out of figure 20 as an example, that would be 29 units and 49 tiles. This means that even if we were to use conversion right now, we would still be left with 20 open tiles, which is once again not the desired effect. To get rid of these 20 open tiles, we would have to scale down our gesture again, until we have 29 tiles. It would not be efficient to first scale up and then scale down, this is why I collect some data to help this process along from the start.
In the following table, you will be able to find the relation between inside units and outside units on the square formation.

Inside units	0	2	3	4	6	9	10	28	41	63
Outside units	5	7	10	12	15	17	20	25	30	35

Table 2: The correlation between inside and outside units on the square formation.

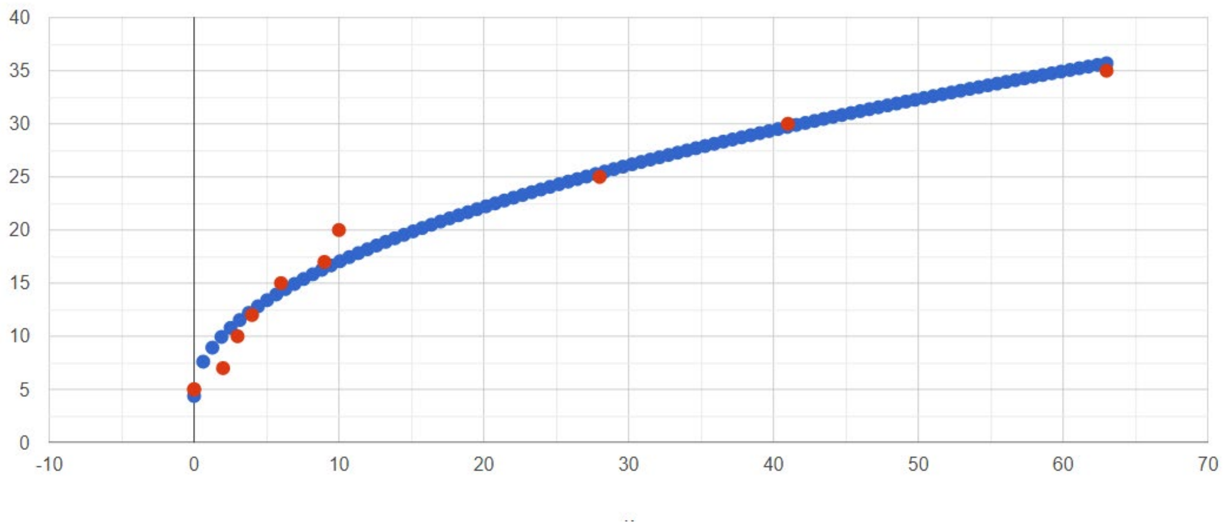

Figure 21: A power curve built up from the statistics in table 2.

$$\boxed{f(x) = 4.38802x0.477923 + 3.86559}$$

Figure 22: The mathematical equation that gave us the power curve in figure 21.

With the equation seen in figure 21, we can calculate the amount of outside units that can at maximum be present when having a certain amount of inside units. E.g., when there are 5 inside units, a maximum of 13 outside units can be present.
If we then take this maximum amount of outside units (13) and extract this from our amount of outside units (24), we get a difference of (+11), meaning that we have 11 outside units too many. Now we can take this difference and divide it by 2 (meeting halfway), which results in (+-6). If we now convert 5 outside units to inside units, we will have 18 outside units and 11 inside units. Now we re-use the equation as described in figure 22 to calculate our scale factor. Giving us a result that can be seen in figure 23 (one-unit underfill).

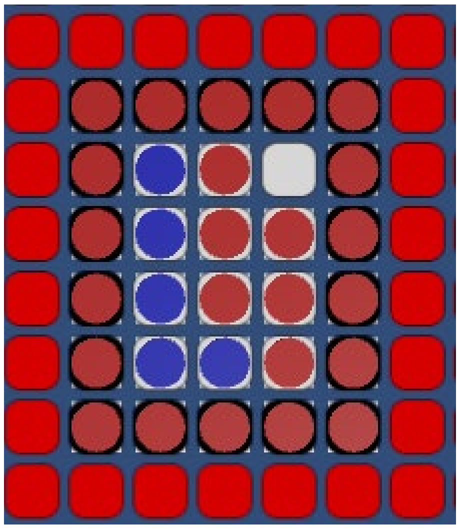

Figure 23: The result of using the new maximum outside units' equation.

The results of our experiment can be explained with the opposite function of the power curve in figure 21. This time we will be looking at the relation between outside units and inside units.

Outside units	5	7	10	12	15	17	20	25	30	35
Inside units	0	2	3	4	6	9	10	28	41	63

Table 3: correlation between outside and inside units on the square formation.

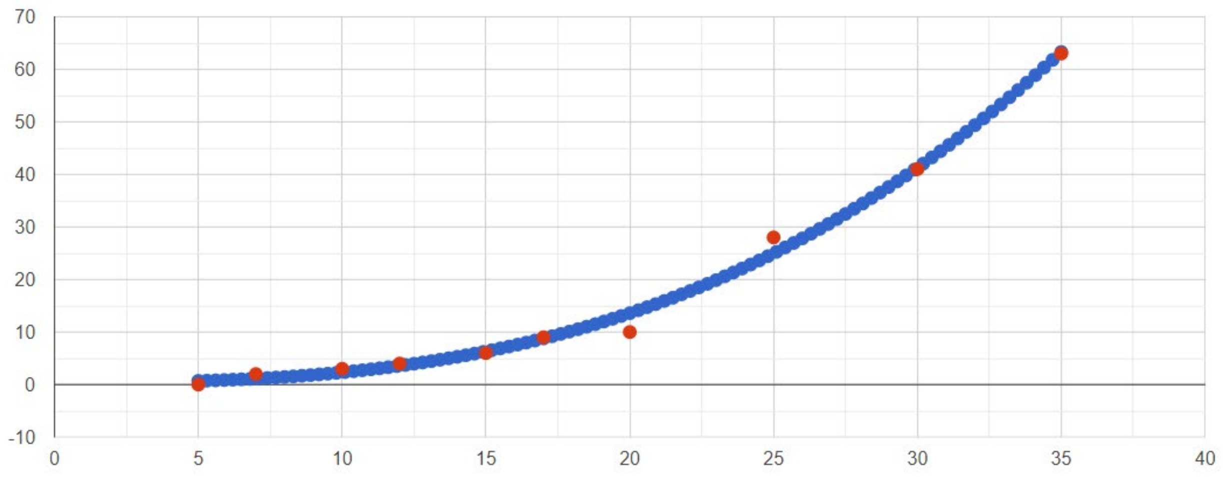

Figure 24: The power curve build up for them statistics in table 3.

$$f(x) = 0.00218871x^{2.894} + 0.638859$$

Figure 25: The mathematical equation that gave us the power curve in figure 24.

If we input our previous result (18 outside units) into this equation, we get (roughly) 10 inside units, one lower than our expected result, thus leading to the underfill in the result of figure 23.

Using this method to convert our outside troops into inside troops can already gives us much better scaling results then before. However, we can still see some undesired results. A very similar, yet opposite problem arises when our inside units are much larger than our outside units, we got an “extreme underfill”. E.g., with 20 inside units and 16 outside units, we can get the following result as can be seen in figure 26.

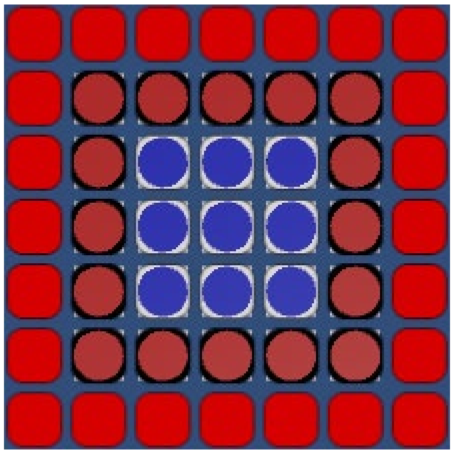

Figure 26: Extreme underfill when inside units are larger than outside units.

To approach this problem, I used the same method for calculating the maximum number of outside units depending on the number of inside units. This time we calculate the maximum number of inside units depending on the number outside units. For our example, 16 outside units, can get us a maximum of (roughly) 9 inside units. We then take our current number of inside units (20) and subtract the maximum number of inside units from this

(9), which gives us 11. Next we divide this number by 2 again (meeting half way) and then convert that number of inside units to outside units, which gives us 16 inside units and 22 outside units. If we then recalculate using the initial scale equation, we get the result as can be seen in figure 27.

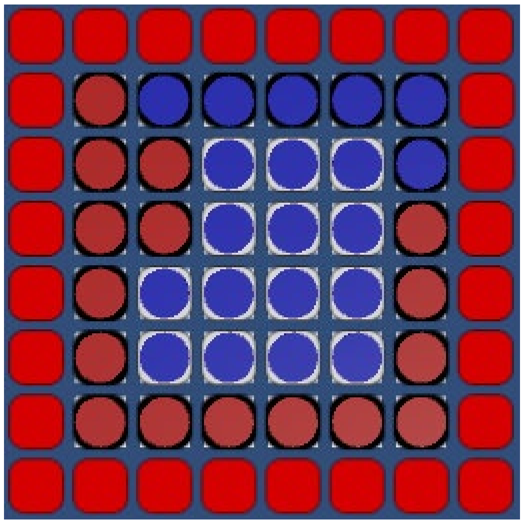

Figure 27: Applying the equation for maximum number of inside units per outside units.

4.2 THE SQUARE PROBLEM

As I had been testing all these equations for scaling mainly on the scare formation, when I started trying out some other equations like the ‘D’, it would not look as good, we would often get an overfill present on our formations (the stacking on top of each other of units). This is why the “square equations” alone were not good enough, but could work as a baseline, as we rarely got cases of underfill.

To enhance our “square equations”, another form of scaling was added.
In this part of the scaling, I would use make use of recursion (redoing a certain action until it was right).

First there is a calculation done to check the number of units we have and the number of tiles we have. If the difference in tiles and units is less than 0, we would be dealing with an overfill. With an overfill, a recalculation of the scaling happened but this time with an added factor (0.01). Next the same check would happen, until we no longer have an over fill.

If the difference in tiles and units is more than 0, we would be dealing with an underfill. With an underfill, a recalculation of the scaling will happen, but this time with a subtracted factor (0.01). Next the same check would happen, until we no longer have an under fill.

With these two checks in place, we would always make sure that the formation on screen had neither an overfill nor an underfill. However, we still ran into a problem, a somewhat common problem with programming, an infinite loop. An infinite loop is when a part of your program gets stuck doing the same thing over and over with no way out, as we were using recursion and constantly checking for no under- and over fill, we could sometimes get stuck in an infinite loop, as some gestures would always either have an under- or over fill for this current number of units. To make sure that we don’t run in to an infinite loop, I made it so that the formation is always allowed to have either 2 units over- or 2 units underfill. This along with a check to see if we are not constantly going back and forward between scaling, made sure that our formations always looked relatively fine.

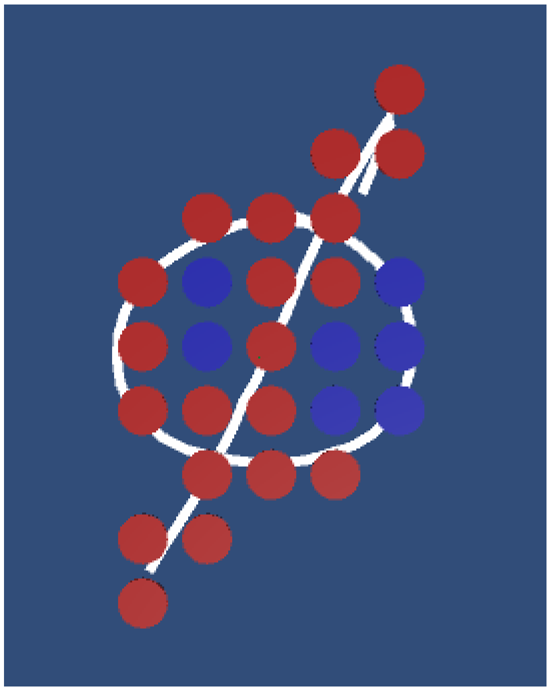

Figure 28: A null symbol formation, with neither over- nor underfill.

5. REAL TIME CHANGES

For the final part of my project, I wanted to implement real time changes in the formation. When a unit in the current formation dies, the formation should be reformed with the new units.

To implement this, I simply added in an option for a unit to die. When a unit dies, they remove themselves from the units in the formation. Next all that had to be done was the formation had to be reformed, but this time without that dead unit. However, since the unit had already removed themselves from the formation, this worked instantly.

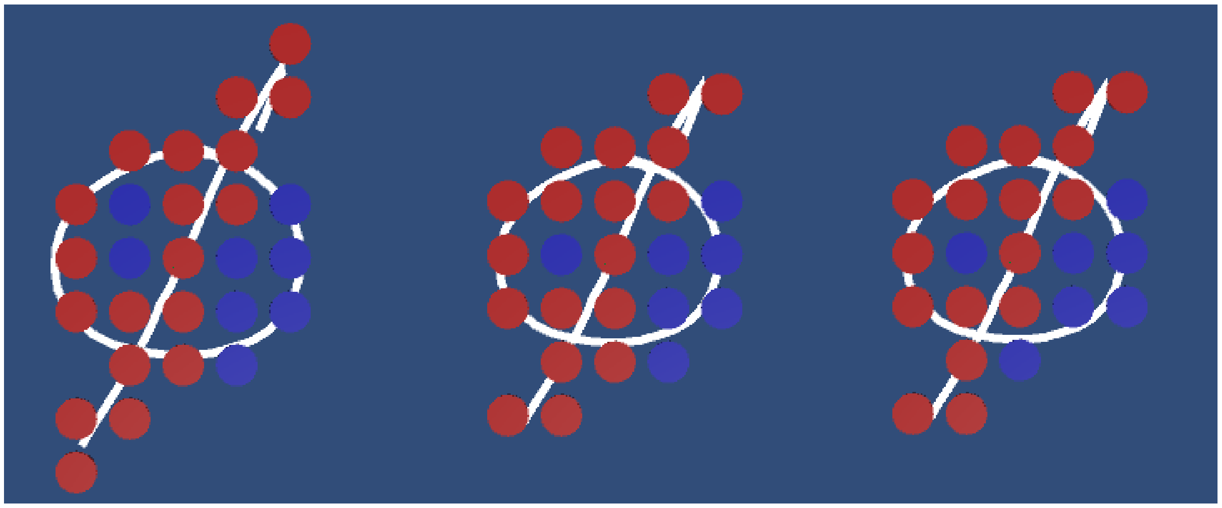

Figure 29.1: Units dying in a null symbol formation.

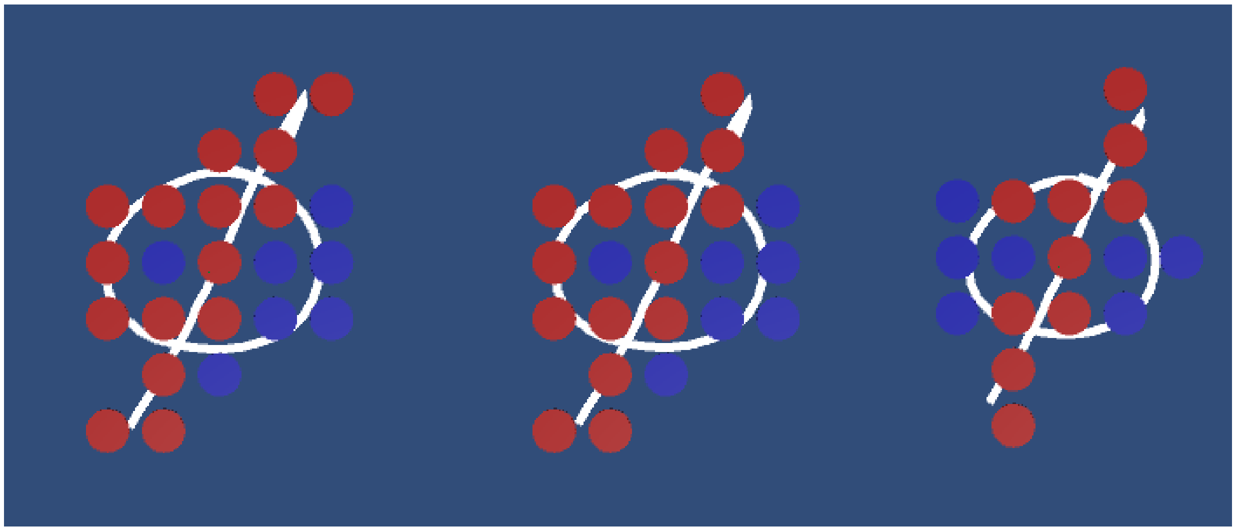

Figure 29.2: Units dying in a null symbol formation.

.

DISCUSSION

The final versions of the project look very good, there is almost never, or barely any under- or over fill, the conversion of the units work very well and (if the unit size is big enough), the formations look close to the gesture that they should represent. There are a few flaws here and there, like how the grid will always more the formation look more robust and squarer, so creating circular formation does not look great. However, it can clearly be seen that by using the grid, the formations lose a lot of their problems that occurred when not using a grid. E.g., the insides of the formation are now also filled in.
The most important result of my findings is the scaling, I think the scaling is a bigger part of this then what I had originally thought, it makes sure that the formation is always well represented, with no unnecessary gaps, as well as the ability to fit any number of units.

CONCLUSION & FUTURE WORK

Can custom sketch based RTS troop formations react and adjust to changes in real time?
Yes, they can. However, it is often in a rather minimal sense, units can die and then the formation will reform, but there are no reactions to other changes, yet.

What could be implemented is a way for the units to react if you take some units from another formation and create a new formation with those units, then both formations should be reformed.

In the results, we can also still see one problem arising, open shapes will never have an inside. This could easily be solved by making sure that the user always draws their shapes closed, but it would be an interesting added feature to for example to let an algorithm close half open shapes.

An added step further is unit and formation movement, in RTS games, you will often have movement by formations, as these formations would like to attack other formations of units. In the current project, movement is not accounted for. However, this could be implemented in the future.

One more thing to consider is optimization, the recognizer in itself is rather optimized. However, the scaling of our formation uses recursion, which can sometimes take a long time to find the right scaling, as we are currently working with a maximum of 50 units, this can still be fast. To create formations of 500, 1000, or even 10000 units, the scaling would be very slow. This can also be added in the future.

BIBLIOGRAPHY

[1] S. Ontañon *et al.*, “A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft,” 2013. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00871001

[2] D. Thalmann, “Crowd Simulation,” in *Encyclopedia of Computer Graphics and Games*, Springer International Publishing, 2016, pp. 1–8. doi: 10.1007/978-3-319-08234-9_69-1.

[3] M. L. Xu, H. Jiang, X. G. Jin, and Z. Deng, “Crowd Simulation and Its Applications: Recent Advances,” *J Comput Sci Technol*, vol. 29, no. 5, pp. 799–811, Sep. 2014, doi: 10.1007/s11390-014-1469-y.

[4] C. W. Reynolds, “Flocks, Herds, and Schools: A Distributed Behavioral Model,” 1987.

[5] Y. ping Chen and Y. yin Lin, “Controlling the movement of crowds in computer graphics by using the mechanism of particle swarm optimization,” *Appl Soft Comput*, vol. 9, no. 3, pp. 1170–1176, Jun. 2009, doi: 10.1016/J.ASOC.2009.03.004.

[6] A. Mamdouh, “Realistic Behavioral Model for Hierarchical Coordinated Movement and Formation Conservation for Real-Time Strategy and War Games,” 2012.

[7] C. Reynolds, “Big fast crowds on PS3,” in *Proceedings - Sandbox Symposium 2006: ACM SIGGRAPH Video Game Symposium, Sandbox ’06*, 2006, pp. 113–121. doi: 10.1145/1183316.1183333.

[8] S. J. Guy, S. Kim, M. C. Lin, and D. Manocha, “Simulating heterogeneous crowd behaviors using personality trait theory,” in *Proceedings - SCA 2011: ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 2011, pp. 43–52. doi: 10.1145/2019406.2019413.

[9] T. Kwon, K. H. Lee, J. Lee, and S. Takahashi, “Group motion editing,” New York, 2008.

[10] Q. Gu and Z. Deng, “Formation Sketching: An Approach to Stylize Groups in Crowd Simulation,” 2011.

[11] W. He, G. Pan, X. Wang, and J. X. Chen, “Real-time crowd formation control in virtual scenes,” *Simul Model Pract Theory*, vol. 122, Jan. 2023, doi: 10.1016/j.simpat.2022.102662.

[12] Y. Ye and P. Nurmi, “Gestimator: Shape and Stroke Similarity Based Gesture Recognition,” New York, Nov. 2015.

[13] J. O. Wobbrock, A. D. Wilson, and Y. Li, “Gestures without Libraries, Toolkits or Training: A $1 Recognizer for User Interface Prototypes,” 2007. [Online]. Available: http://www.opera.com/products/desktop/mouse/

[14] L.-P. Morency, D. Bohus, H. K. Aghajan, SIGCHI (Group : U.S.), Association for Computing Machinery, and ACM Digital Library., *Gestures as Point Clouds: A $P Recognizer for User Interface Prototypes*. 2012.

[15] L. Anthony and J. O. Wobbrock, “A Lightweight Multistroke Recognizer for User Interface Prototypes,” 2010. [Online]. Available: http://depts.washington.edu/aimgroup/proj/dollar/ndollar.html

[16] K. P. Fishkin and B. A. Barsky, “An Analysis and Algorithm for Filling Propagation,” in *Computer-Generated Images*, Tokyo: Springer Japan, 1985, pp. 56–76. doi: 10.1007/978-4-431-68033-8_6.

[17] da Viking Code, “PDollar Point-Cloud Gesture Recognizer.” 2015.

APPENDICES