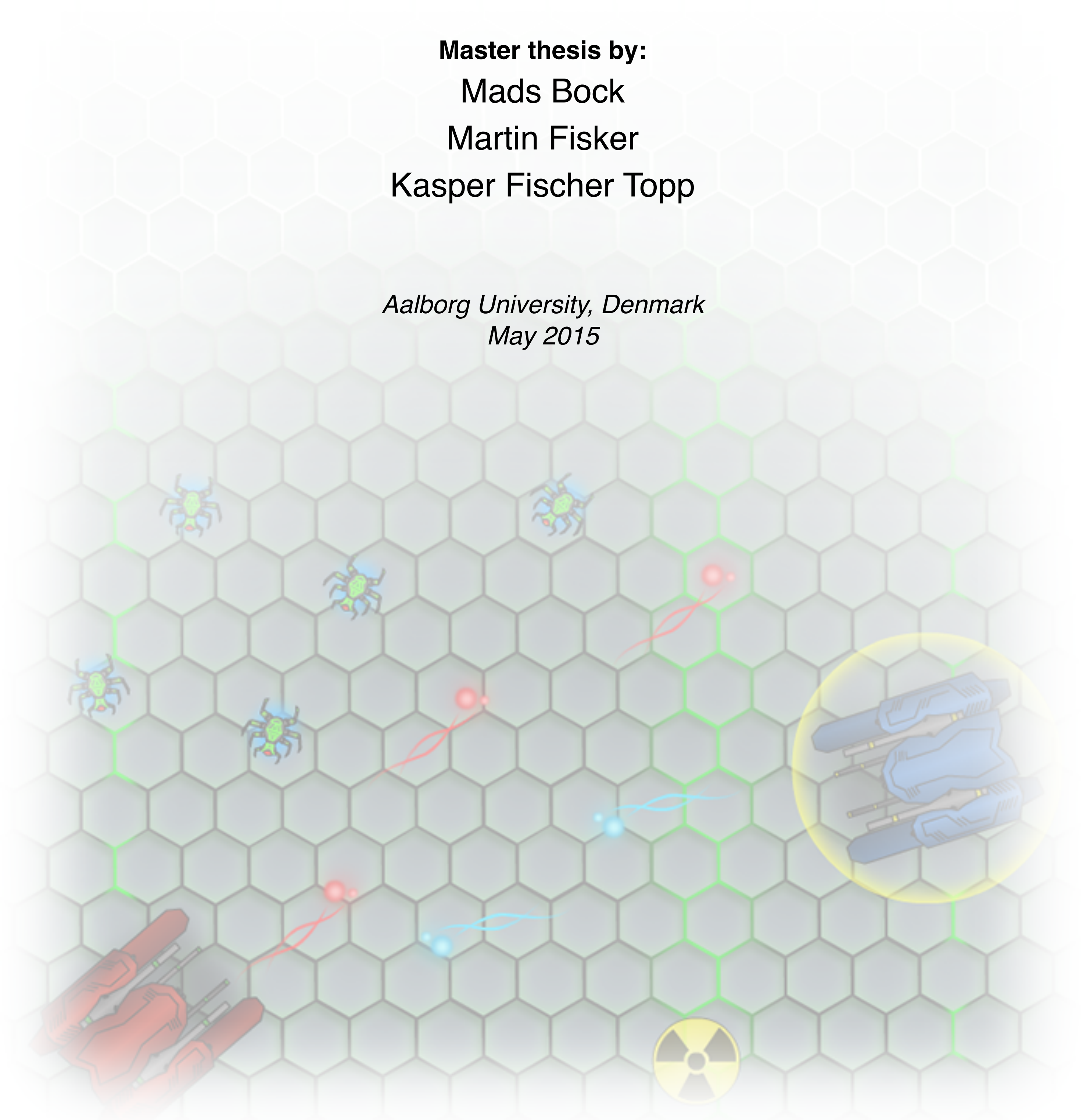
Tangible Widgets for a Multiplayer Tablet Game in Comparison to Finger Touch

Master thesis by:
Mads Bock
Martin Fisker
Kasper Fischer Topp

Aalborg University, Denmark
May 2015



Abstract

This master thesis investigates the use of tangible widgets in tablet games. Tangible widgets are small physical objects that can be recognized by the touch screen. Assumed advantages of tangible widgets are that they increase the intuitiveness of rotating objects on the screen, and they better afford user identification, when compared with regular finger touch. It is assumed that these advantages will make them suitable for tablet games. This master thesis details the design process of implementing a tablet game using tangible widgets, called Hover Wars. Using this game, a user test was conducted in order to explore user's reception of this new form of interaction. The data from the test showed that a majority of users found it easier and more intuitive to use widgets than finger touch. However it also reveals that widget size, and occlusion of the screen, are potential issues when using widgets, and should be taken into account when designing them.

Introduction

It has been suggested that tangible widgets can be a valid interaction method for use in tablet games (Bock et al., 2014). Tangible widgets refer to the use of small graspable objects to interact with objects on a touch screen, instead of the more regular finger touch interaction. There are several studies that have looked into tangible widgets, though very few of them deal with widgets specifically as an interaction method for use in games. It is assumed that tangible widgets afford better immersion, and is more intuitive, in comparison to finger touch, due to tangible feedback and a closer relation to physical objects. Therefore it could be interesting to look at tangible widgets as an interaction method for video games and see how it changes the user experience. This project will investigate the use of tangible widgets as an interaction method in a tablet game, and try to discover which advantages and drawbacks there can be when using tangible widgets. To do this, we will design and implement a system for detecting tangible widgets, and use this system to implement a game. The results will be gathered through a user test.

This report is split into 9 chapters, including this introduction. After this, the “Related Works” chapter will introduce the different studies that inspired this thesis and looked into similar areas. The “General Widget Design” chapter will talk about our widgets and how they are designed. “Physical Widget Design” will explain how the physical widgets were produced. “Widget Detection Algorithm” will detail the design and implementation of the detection algorithm that will make it possible to use the widgets with the tablets. And the “Game Design” chapter will talk about the game design. After that, the “Experiment” chapter will detail the setup and execution of the different user tests that were made as part of this thesis. The results of these tests will be listed in the “Result and Discussion” chapter, and finally the “Conclusion” will sum up the different chapters and talk about what future work can be done in this field.

Table of Contents

1 Related Works

In this chapter we will summarize similar studies and other relevant works that has inspired this project. We briefly describe each study and its relevance to this project.

Bock et al. 2014

Bock et al. (2014) investigated the use of widgets for tablet games. Their experiment was designed to test whether widgets were better suited for fast-paced, or slow-paced games. They did this by implementing two different games, a fast-paced and a slow-paced one, and having users test both games, with both widget- and touch control schemes. The users would then report what control scheme they preferred in a series of different questions (e.g. “Which method was the funniest”, and “Which method was most precise”). The results were not significantly in the favor of using widgets. Yet their paper concludes that there were ground for looking further into the use of tangible widgets for computer games.

There were a couple of issues with Bock et al. First off, they claimed to have a lot of detection errors when using widgets in their test, which might have led to the poor preference results. Also, it is worth noting that their widgets used only three touch points for detection. They bring up that their algorithm needs updating, but they do not talk about updating their widget design. In order to improve upon their results, this project will focus on a new way to design the widgets that will increase their usefulness.

Schaper 2013

Schaper (2013) introduced a way to design widgets using three or more points. The study was an attempt to list the design constraints that must be fulfilled in the design of tangible widgets. His design of widgets was the main inspiration for the widgets designed in this project. However Schaper had large focus on the design of the widgets, but doesn’t really touch on the use of widgets in a real context, like video games. We plan to expand upon his design of widgets and see how well they function as a means of interaction in tablet games.

Voelker et al. 2013

In 2013, Voelker et al. introduced what they called Passive Untouched Capacitive Widgets (PUCs). By designing the widgets in a specific size, with a specific pattern of touch points, they claimed to be able to create widgets that could be detected untouched on any capacitive screen, without the use of batteries or any other power source. Having widgets that work untouched would allow for even better interactions with the screen, as the device would be able to distinguish whether a widget is removed from the screen, or simply not touched anymore. Unfortunately, we were unable to reliably replicate their results on our iPad, and therefore it was decided that we could not rely on having untouched widgets in our game design.

Buxton 1990

Buxton (1990) introduced a three-state model of interaction that can be used to describe the affordances of different modes of interaction (e.g. mouse, joystick, touch-screen etc.). He identifies three different states. State 0 is known as the Out of Range (OOR) state, where the device is not affected by the actions of a user (e.g. a finger hovering over, but not touching, a touch-screen). State 1 is known as the Tracking state, where the device follows the movements of the user, but is not interacting with it (e.g. moving a mouse around, without pressing the button). And finally State 2, known as the Dragging state, where a device is aware, and is manipulated by the user (e.g. Dragging an icon with a mouse, where the button is pressed).

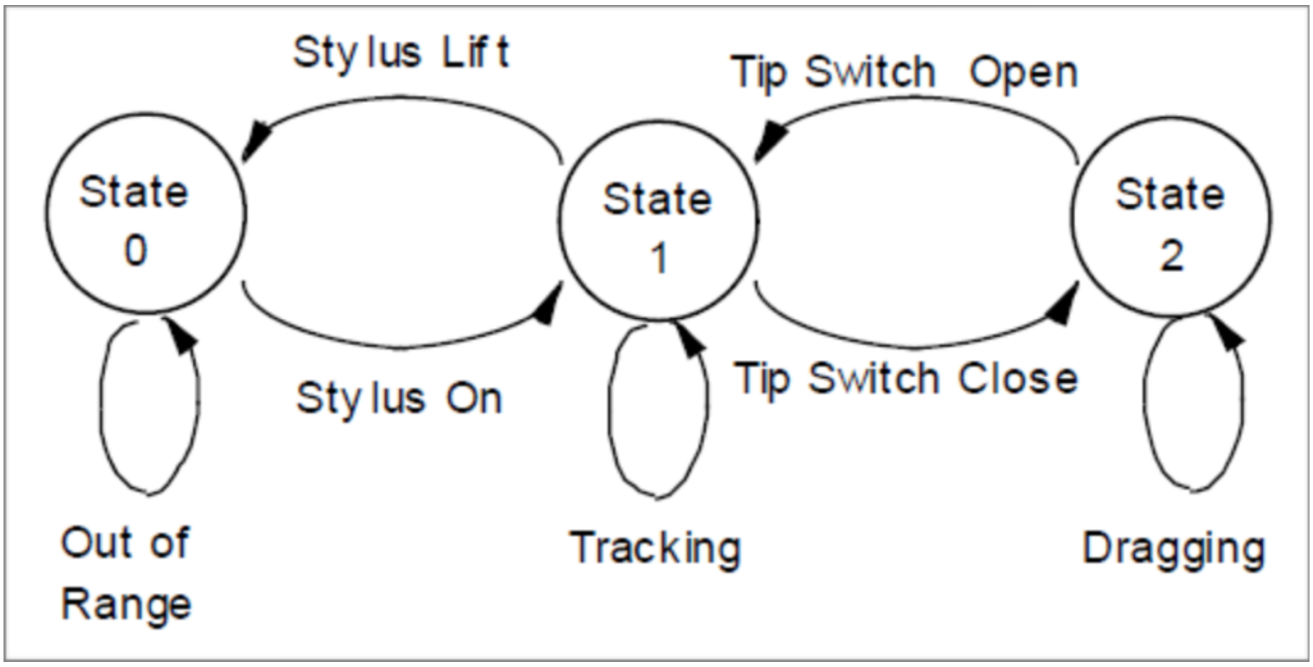


Figure 1: Buxton's Three-State model of interaction. In this case it is using a stylus as an example device.

Buxton states that not all modes of interaction have all three states. For example a mouse only has state 1 and 2 (lifting the mouse makes no difference after all), while a capacitive touch screen only has State 0 and State 1.

The tangible widgets used in this project will have state 0 and 1. State 0 will represent the widget not being on the screen, and state 1 will represent putting the widget on the touch-screen and moving it around. Due to the widgets not being untouched, state 0 will also represent the widget being on the screen, but not being held.

Fitzmaurice et al. 1995

Fitzmaurice et al. (1995) are considered the first to academically introduce the concept of tangible widgets. In their paper they refer to them as "Graspable User Interfaces" According to them, the philosophy of Graspable User Interfaces has the following advantages:

- "It encourages two handed interactions"
- "Shifts to more specialized, context sensitive input devices"
- "Allows for more parallel input specification by the user, thereby improving the expressiveness or the communication capacity with the computer"
- "Leverages off of [people's] well developed, everyday skills of prehensile behaviors for physical object manipulations"
- "Externalizes traditionally internal computer representations"
- "Facilitates interactions by making interface elements more "direct" and more "manipulable" by using physical artifacts"
- "Takes advantage of our keen spatial reasoning skills"
- "Offers a space multiplex design with a one to one mapping between control and controller

- “Affords multi-person, collaborative use”

These are some of the advantages tangible widgets could add to the user experience in a game.

Yu et al. 2010

Yu et al. (2010) in their study talks about Spatial Tags and Frequency Tags as two different ways of facilitating user identification in Tangible User Interfaces (i.e. an interface that uses Tangible Objects or Widgets). Spatial Tags “uses multi-point, geometric patterns to encode object IDs” (Yu et al. (2010)). Frequency Tags “simulates high-frequency touches in the time domain to encode object IDs” (Yu et al. (2010)). The drawbacks of using Spatial Tags is that it requires several touch points per object, where Frequency Tags potentially only requires one touch point. Also, the mechanism used in touch devices would merge close touch points, further limiting the number of different widgets that could be made. Conversely the number of potential Frequency Tags that can be made are limited by the response rate of the given touch device.

We chose to go with spatial tags in this project, for simplicity reasons. The iPad that we use for this project can detect 11 different touch points, which for our purposes will be enough to implement.

Chan et al. 2012

Chan et al. (2012) demonstrated how it was possible to create stackable widgets, that could communicate to the touch device the number of widgets in a given stack. The CapStones as they named them could communicate with each other electrically when stacked in other to “hand down” the number of widgets to the underlying widgets and ultimately to the touch device.

While the idea is good, and their implementation shows to work, there are two major problems with CapStones. First, the CapStones could only be stacked to a certain height, before no longer being able communicate said height to the device, limiting the ways the CapStones can be used in an application. Furthermore, the CapStones only works when they are stacked completely on top of each other, and would therefore not worked if for instance the stones were stacked crooked. But it goes to show how some people are investigating the use of these tangible widgets.

Blagojevic et al. 2012

Blagojevic et al. (2012) used a version of Tangible Widgets to create tools for creating what they describe as “an innovative drawing application for use on Apple iPads and Windows tablets with the ability to detect tangible drawing instruments” (Blagojevic et al. (2012). This is another interesting application of Tangible Widgets.

Kratz et al. 2011

Kratz et al. (2011) made a study of using Tangible Widgets like the ones used in this project. Their study showed that people were not automatically preferred widgets, and that widgets not necessarily improved performance. They call for more research into the design of widgets in order to validate their properties.

Disney AppMATes

Disney describes AppMATes as: “Mobile Application Toys and the first physical toys that digitally interacts and magically come to life on an iPad”. These are basically toy cars with a series of points on the bottom that can be recognized by an iPad's touch screen. Placed on the touch screen the AppMATes cars can be used to drive around a virtual world shown on the iPad. The advances mentioned about of the toys are that they do not require any form of electricity or wireless connection; it is simply downloading the app and placing the AppMATes car on the screen to play.

The AppMATes cars were used as a finished product inspiration for the widgets made for this project, as it was the closest commercial product to what we had in mind for our widgets. The ideas of needing nothing extra to play but putting the widget on the screen and design choices with regards to shape and size were some of the things taken away from examining the AppMATes product.

2 General Widget Design

The general idea of the tangible widgets is to make a handheld real world object that can be placed on a tablet's touch screen in order to interact with the virtual environment on the tablet. In this case a game where the widget is used to control a specific virtual game object and move it around the screen and interact with the world using the widget. For this purpose the widget needs to communicate its position and orientation to the tablet as well as a unique id so it can be differentiated from other potential widgets on the screen. To achieve this, a widget system has been designed with inspiration from Schaper (2013) that uses a number of different touch points on the bottom of a widget to calculate these variables.

2.1 Touch Point System

The system consists of three kinds of touch points; the main detection points, identification points and interaction points (Figure 2).

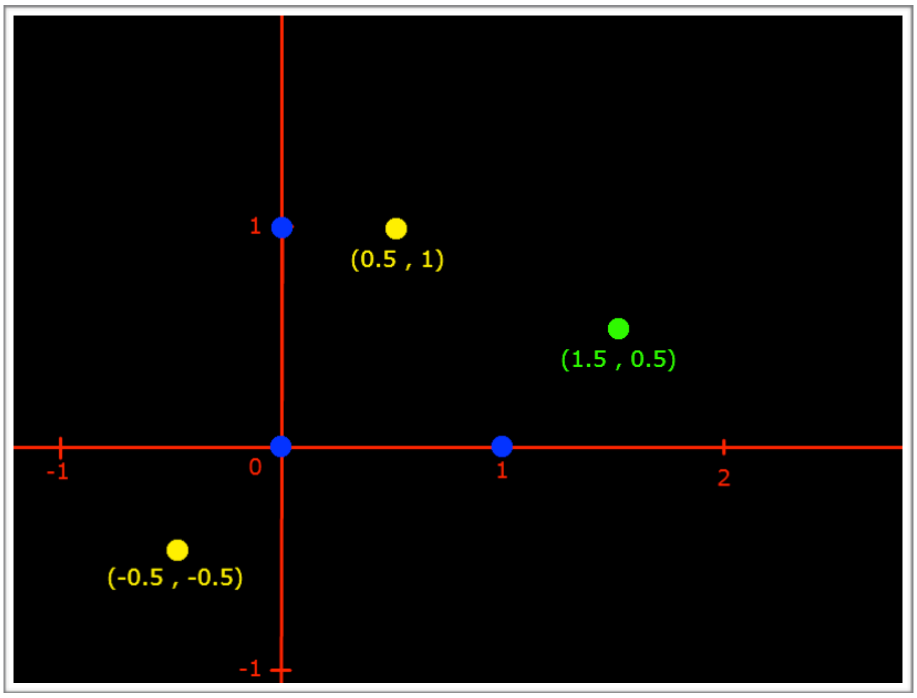


Figure 2: Example of recognition system with detection points (blue), the coordinate system (red) generated by the detection points and interaction points (yellow) and identification point (green) set into the coordinate system with their coordinate sets shown.

The detection points are three touch points that are placed in a right Isosceles triangle and used to detect the presence of a widget dependent on how close the angle is to 90 degrees and how equal the length of two sides are. The triangle is used to detect the position of the widget and its rotation using the triangle's orientation. These three points are then also used to construct a 2D coordinate system with origin in the right angle point and the lines out to the two remaining detection points as the x and y axes. This 2D coordinate system is referred to as the widget coordinate system.

The identification point of a widget is a specific set of coordinates in the widget coordinate system, that is used to identify it and tell it apart from other widgets. The interaction point of a widget is a set of coordinates assigned to a widget as a button. The interaction point only exists when the corresponding button is being pressed. Interaction points can have the same position and coordinate set on all widgets even if the buttons have different functions for each widget, as the id can be used to distinguish between these.

A limitation of this system is that touch points other than the detection points could not be placed in such a way that any combination of touch points formed another right Isosceles triangle. This would by the system be detected as another widget, resources would be spent on trying to extract information from it and errors could occur in detecting the actual intended widget.

3 Physical Widget Design

This chapter first describes the physical design and construction of the widgets made during this project. Then there is also a rough description of the different prototypes that were developed during the project, what things they were used to investigate and what were learned from them.

3.1 Final Widget Design

This section goes through the design of the widgets used for testing (Figure 3 and 4), their individual parts and the different considerations made when deciding on how to design and assemble them.

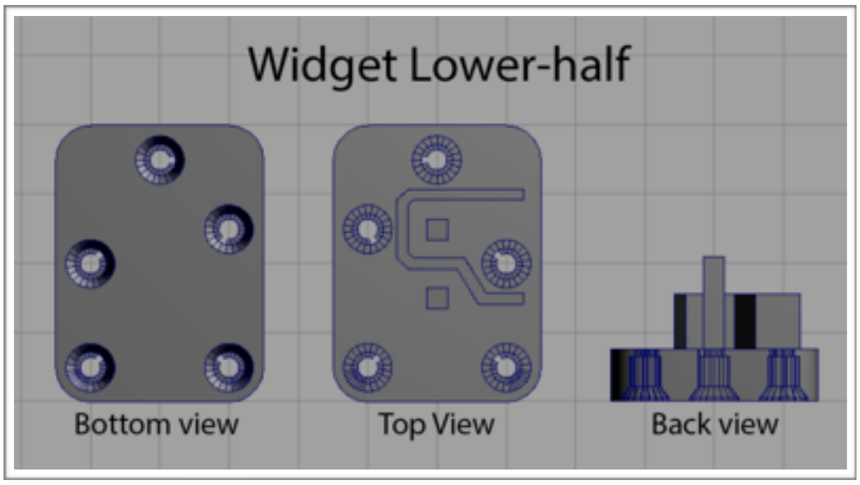


Figure 3: Model of the lower half of a widget seen from three different angles with the holes, support columns and divider outlined.

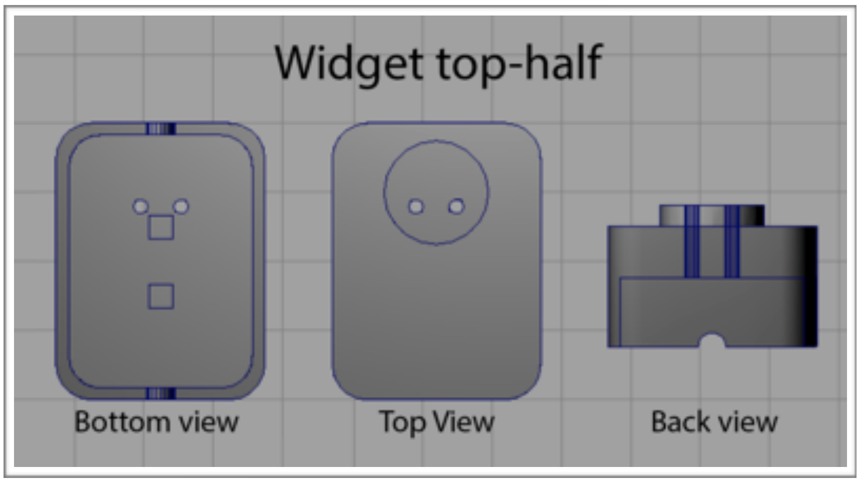


Figure 4: Model of the upper half of a widget seen from three different angles with holes, platform and gaps outlined.

Size and Shape

The size and shape of the final widgets was mainly chosen with 2 things in mind. First was the user interaction with the widget; being able to use it easily and without too much trouble for a longer period being the main concern. The widgets had to be comfortable and easy to hold on to while the users moved and turned quickly across the tablet screen in accordance with the fast pace of the game. The button of the widget also had to be easy to reach and identify to facilitate users being able to quickly press it when needed.

A general square shape was chosen to make it easy for the user to place two fingers on opposite sides of the widget to get a firm grip (Figure 5) as opposed to e.g. a circular shape. While also being able to accurately point the widget in the desired direction by pushing on it with one of the fingers. The button was placed on top of the widget and raised on a circular platform to both make it easily noticed and to keep it away from the sides where the widget is held. The button was also placed at the front of the widget to help indicate which way to turn the widget and also so the finger pressing it does not need to bend too much to reach it.

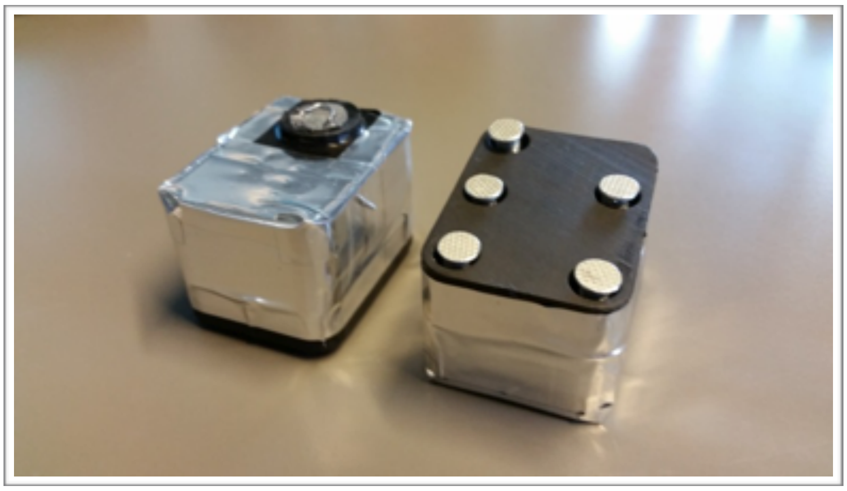


Figure 5: Finished widgets used for testing pictured from the back where the user would hold them.

The second main consideration was how well the widgets integrated with the gameplay on the tablet. The more important thing here was the widget size compared to the screen size of the tablet. The bigger the widget the less screen area there would be for the users to move the widget around on lowering the possible interactions with the game, possibly making them feel somewhat “trapped” on too small a screen. The height of the widget was also a factor here as the higher it was, the greater an area of the screen directly in front of the widget would be obscured (Figure 6).

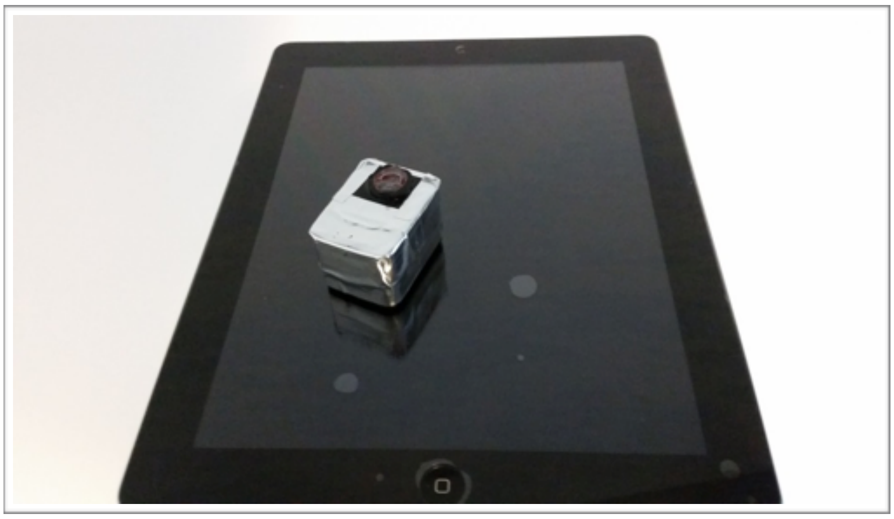


Figure 6: Widget shown on tablet screen approximately from users point of view.

To minimize the amount of the screen that would be obscured by the widgets, the width and length of the widget parallel to the tablet screen, was made to cover the least amount of area possible.

The limiting factors here was the touch points had to have a certain size to be recognized and had to be a minimal distance from each other to be differentiable to the touch screen.

This minimization of the widgets could come into conflict with the first consideration of making the widget comfortable to hold and use. To resolve this conflict the height had to be large enough to allow comfortable room for a finger or two on the side but low enough to only block a small part of the screen in front of it. While taking these things into account to help determine a good size for the widgets, inspiration was also taken from the Cars 2 AppMATes ("AppMATes" 2013) cars as the general volume of our widgets would be approximately similar to these.

Position of Touch Points on Widget

The touch points at the bottom of the widgets was, as mentioned, positioned to minimize the surface area that the widget covered. There are a couple of constraints that has to be taken into account when doing this. Firstly in order for a touch point to be detected by the tablet at all it has to be a minimum size and secondly the touch points must have a minimum distance between them or the tablet could read them as being the same object (Figure 7).

With this in mind the three detection points, which must form an isosceles right triangle, have been placed in the corner of the widget to give as much room as possible to the remaining touch points. This is important because they need to be spaced far enough apart in the grid of the detection points to minimize identification error.

The remaining touch points had to be placed in the grid of the detection points so they each had a unique set of x and y coordinates in order to tell them apart from each other. Their placement was further limited because none of them could be placed in such a way that any combination of three touch points formed another isosceles right triangle, as this could be interpreted as another widget by the detection algorithm.

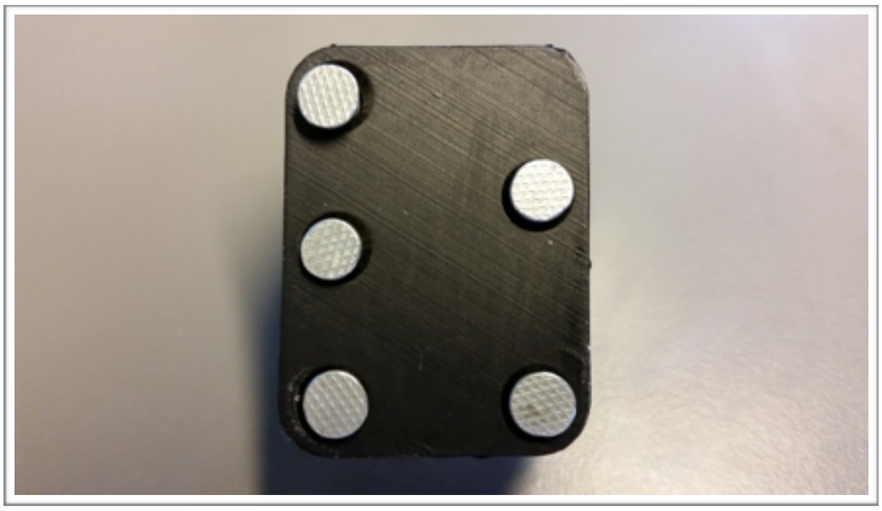


Figure 7: Touch points at the bottom of the widget.

The button touch point was placed opposite the right angle detection touch point, but lowered down some to avoid making three new possible triangles identical to the detection triangle and to allow more space above for the identification touch points. The identification touch points were placed above the rest to have enough distance to allow the tablet to differentiate them from the other touch points while also having distinct enough grid coordinates to avoid misidentification of the widgets.

Nails Used as Touch Points

A conductive material was needed to make the touch points recognizable by the tablet screen and as mentioned they also needed to have a minimum size or the tablet would simple filter them out. Nails worked well for this as the head can give the necessary circular area to be recognized by the screen when held against it. The body of the nail then goes through the widget helping to keep the touch points stable by limiting the heads vertical movement and also makes it easier to connect the touch points with each other inside the widget.

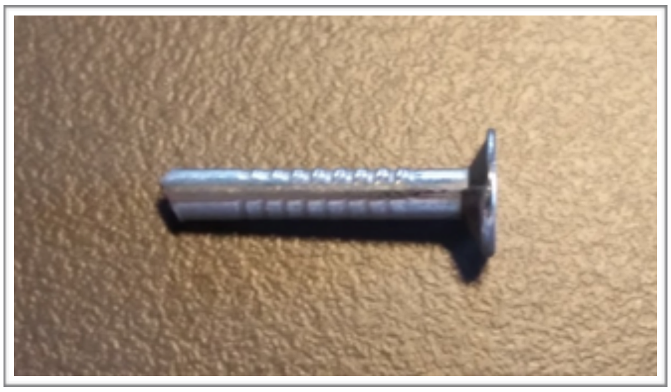


Figure 8: Nails used as touch point in the widgets.

Springs Leading Capacitance

In the holes of the nails in the widget, springs (Figure 9) are placed in order to help keep the nail heads aligned with the screen. The springs coil around the nail bodies and the ends of the springs are resting against the back of the nail heads and around the hole where the nails enter the interior of the widget. The springs are important because if the nails are stationary the smallest tipping of the widget could cause the touch point to leave the screen and the widget would not be detected properly. Even if the nails where perfectly aligned this would be a problem, but the springs resolve this by pushing the nails out a little if the widget is tilted slightly so the nail still touches the surface of the screen. In order to keep the nails from dropping out of the widget a blocker must be attached to the part of the nail body that is in the interior of the widget, in this case it is a small piece of strong tape that is too wide to go through the nails hole.

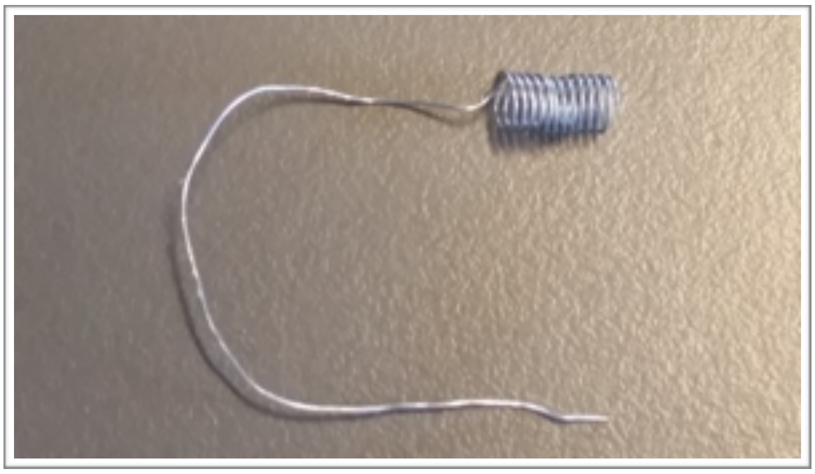


Figure 9: Springs with extra length of wire used in widget.

Connecting the touch point nails with each other was done through the springs to let the nails move more freely when pressure was applied to them. The end of the spring coil towards the interior of the widget continued in a straight uncoiled steel wire that went through a small gap on the edge of the nail holes. This way there are not any direct connections between the nails, preventing them from affecting each other's positions by pulling slightly on each other via a wire when they move up and down (Figure 10).

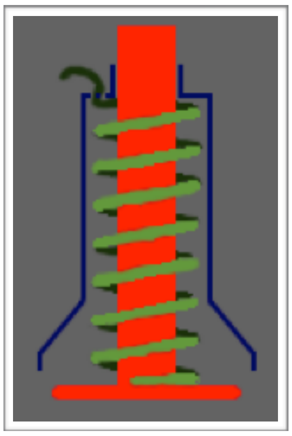


Figure 10: Sketch of a nail (red) and a spring (green) in a hole (blue) in the widget.

Interior of the Widget

The straight part of the spring coils continue in the interior of the widget (Figure 11). Here the three detection touch points and the identification touch point's wires are led to a central pillar on one side of a divide and wrapped around the pillar to connect them with each other. A wire is connected to those around the pillar and it is led to the exterior of the widget through a hole in the side. In the interior on the other side of the divide, to keep it separate from the other wires, the wire from the button touch point is coiled once around another central pillar. The wire is led through a hole in the top of the widget emerging on the button platform on the outside of the widget and pushed into a hole to secure the wire to the platform.

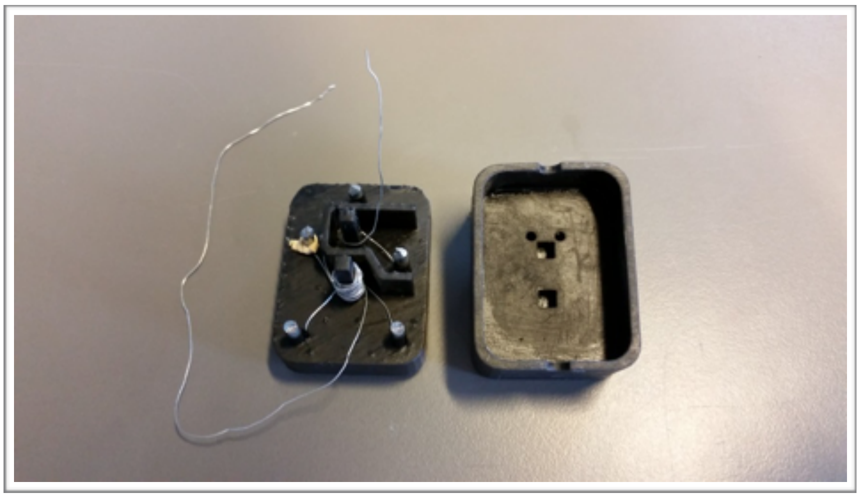


Figure 11: Interior view of an assembled widget with the top taken off.

Exterior of the Widget

The exterior of the widgets are entirely covered in a layer of insulating tape on top of which the wires are laid (Figure 12). The button wire contained to the button platform and the detection and identification wire wrapped once around the sides of the widget and then once over the top from one side of the widget to the other. Touching the respective wires on the outside of the widget will lead a person's capacitance through the wires to the springs and nails to the corresponding touch points on the bottom of the widget allowing the tablet screen to detect them.

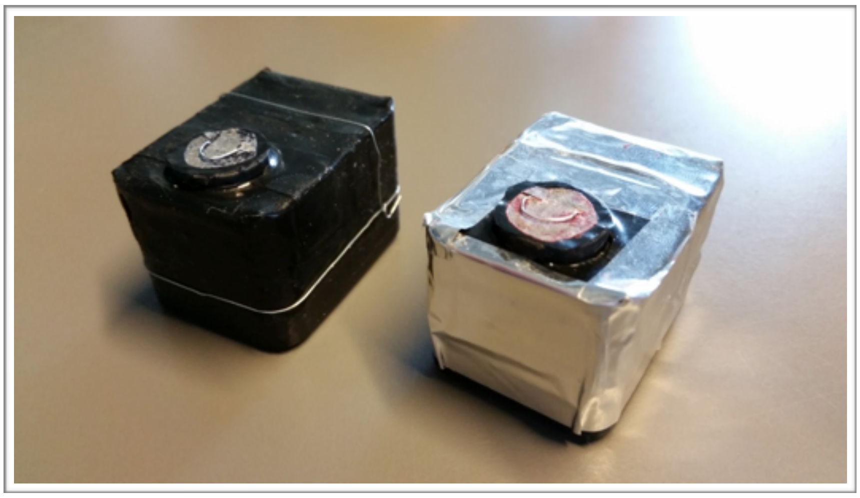


Figure 12: Exterior view of the widgets; the right one with only insulating tape and wires put on, the left one with the metallic tape added.

To make sure the user can touch the widget anywhere on the exterior but the button platform and still have the widget be detected it a layer of metallic tape is put on top of the insulating tape and the wires (Figure 12). This is tape with metal imbedded in it which makes it conductive, allowing it to carry a charge and convey the capacitance of a person touching it. The layer of insulating tape is used in order to prevent any charge built up in the metallic tape from interfering with the wires on the inside of the widget.

3.2 Prototypes

Reaching the design used for the testing in our project required several design iterations and a number of prototypes were constructed to test the validity of the different designs and assembly ideas and methods. This section will go through the prototypes in a chronological fashion starting with the previous widget design to the final prototypes made before constructing the widgets used for testing.

Previous Widget Design

The basis for constructing the new widgets for this project was the widgets that were made for Bock et al. (2014). These where made with three touch points on the bottom forming a unique triangle used for detection and identification (Figure 13). The touch points were made with nails pushed through the bottom of a base where the head of the nails acted as touch point contacts with the screen. The end of the nails where linked inside the widget and led to the outer sides where a strip of aluminium foil was wrapped around the base to convey a person's capacitance to the nail head. On top of the base a figure resembling the in-game object the widget controlled was placed.

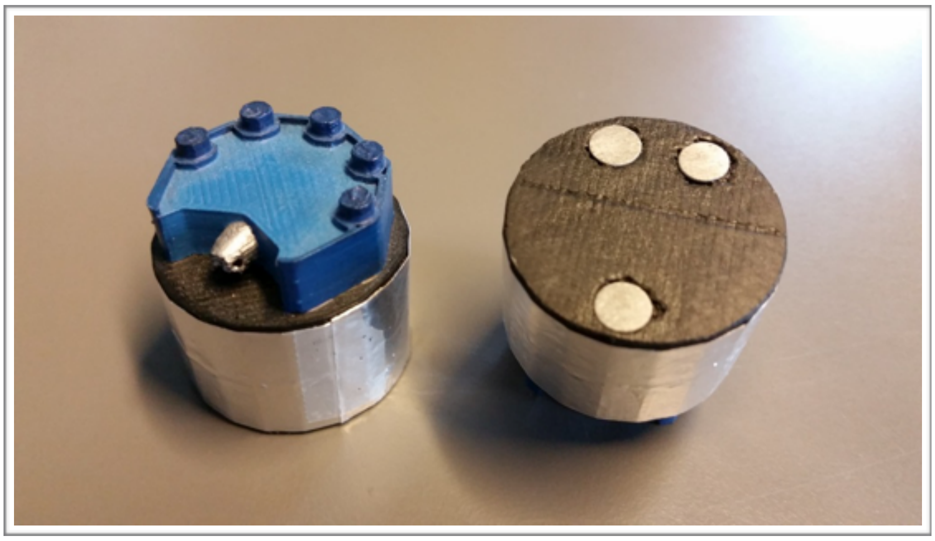


Figure 13: Widget from previous project shown standing up (right) and turned to view the bottom (left).

The detection method with regards to the number of touch points and how to use them to detect and identify the widgets changed significantly from the onset of this project. The general ideas behind the making of the widgets and the lessons learned building them could still be used in this project. Some weaknesses in the design still needed to be addressed, such as losing touch points from the screen with even minor tilting of the widget.

Prototype 1

The first prototype was made mainly as a way to test if the new detection algorithm worked and if so, what amount of precision the algorithm could use and still reliably detect the widgets. The prototype was also used to see what kind of precision could be expected from the touch screen's reading of the physical widgets. In regards to widget touch points being a consistent distance from each other when moving and how often a touch point was not detected by the screen.

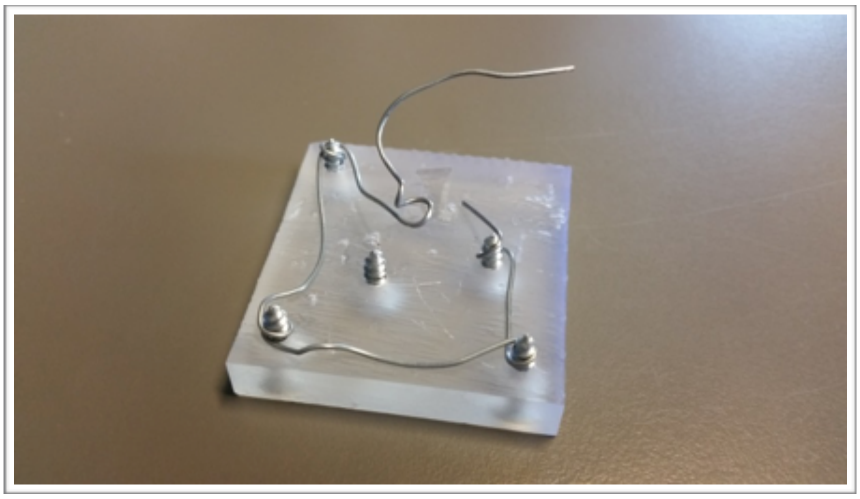


Figure 14: First prototype made to test detection algorithm.

The prototype is simply a square piece of plastic with five holes in it placed to correspond to the pattern needed for the detection algorithm the find and identify the widget and use the button function. Screws are screwed into the bottom of the widget and connected with a piece of wire on the other side. The screw heads then work as touch points when touching the wire or the individual screw in case of the button touch point.

Prototype 2

The next prototype was made with the intention to get the actual size of the widget right as was discussed in a previous part of this section. The prototype (Figure 15) was used to see if the minimization of the widget could still be detected be the algorithm and how reliable it was. This version was basically the same as the previous prototype simply with the touch points placed in a different and tighter configuration.

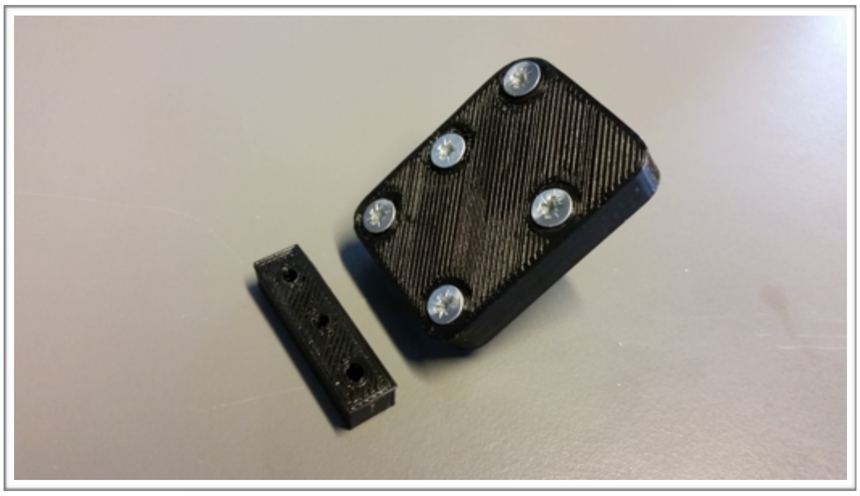


Figure 15: Second prototype made with a hole testing piece in front of it.

In conjunction with this prototype some smaller plastic square (Figure 15) with different sized holes were made to find the best fit for the screws used. This was in an effort to allow the screws to easily be screwed in and out to align all the screw heads perfectly with the touch screen.

Prototype 3

This prototype was made to find an alternative to the stationary screws as the system was still too susceptible to detection loss if the widget was tilted a little bit on the touch screen. The first idea was to let a nail sit loosely in the hole so gravity would let it fall down and touch the screen when the widget was placed on it. The prototype was made with three different holes of varying diameter to see if this would work and a fourth hole where a spring could be placed between the nail head and the hole bottom in case gravity was not sufficient (Figure 16). The loose nail system did not work that well as either the hole was too tight not allowing free movement of the nail or it was so loose the nail could move around enough to be off the touch screen and undetectable by it.

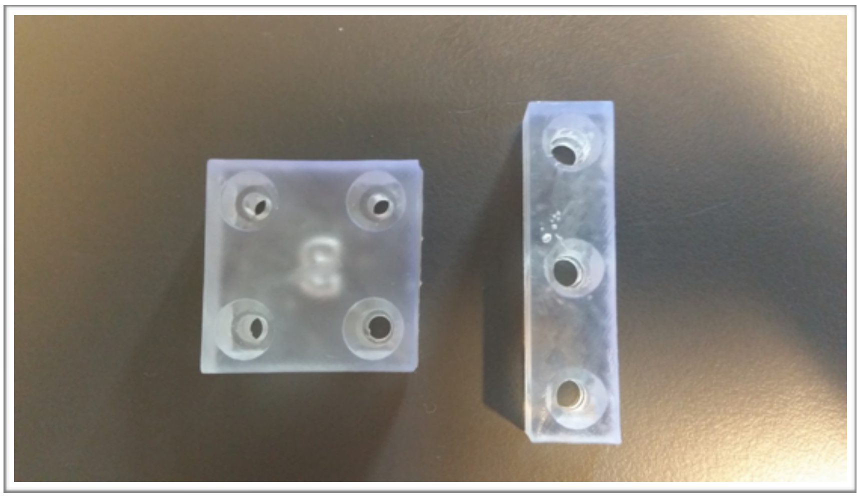


Figure 16: (Left) square with three different sized holes for loose nails and one for a nail with spring (lower right hole). (Right) rectangle with three different sized holes for nails with springs.

The touch point using the spring worked much better and had almost constant contact with the screen with only very few detection losses when moving the widget. An extra square with more spring holes of different diameters was made to find the ideal hole size (Figure 16). This turned out to be a tighter fitting hole with just a little bit of room for the nail to move from side to side, so the nail head could right itself against the screen if the nail was not entirely straight.

Prototype 4

The next prototype was a widget with the intended size and shape using the spring system for touch points (Figure 17) and with a top that could be used to make a finished usable widget (Figure 18). This prototype was used mainly to find a good strength and length for the springs so the widget was detected properly without needing to be pushed too hard against the screen.

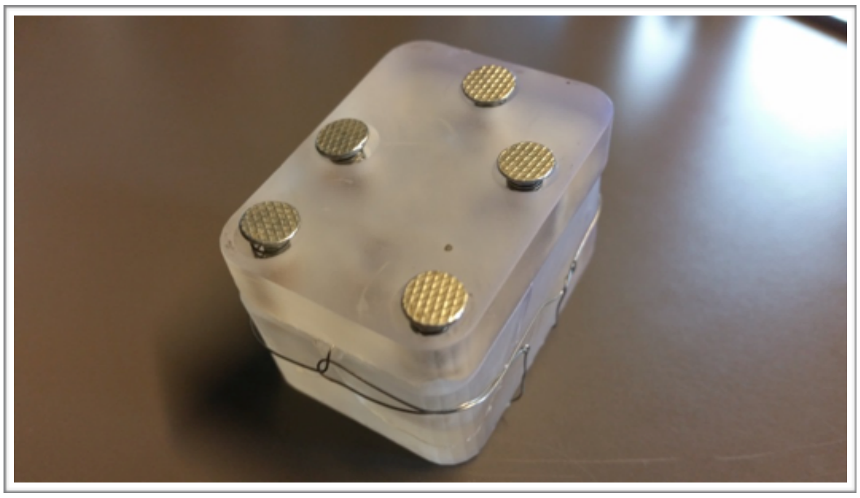


Figure 17: Fourth prototype viewed from the bottom with the nail and spring system added.

Finishing the widget by connecting the touch point nails suitably and making a connection to the exterior of the widget for leading user capacitance to the nails was also investigated. Copper wires were tried with one end attached to the nail, then connected together inside the widget and lead to the outside. This did not work very well as the wires somehow made a charge so the screen could detect the touch points even when a person was not touching the widget or wires themselves. This might not be a bad thing with the detection and identification points, although the detection of touch points was a bit too sporadic to be used in this case. The button however does not work if its touch point is detected when not touching the button so the copper wires were scrapped. A solid thin steel wire was chosen instead as it did not create these sporadic detection problems, even if it was a little harder to attach properly to the nails.

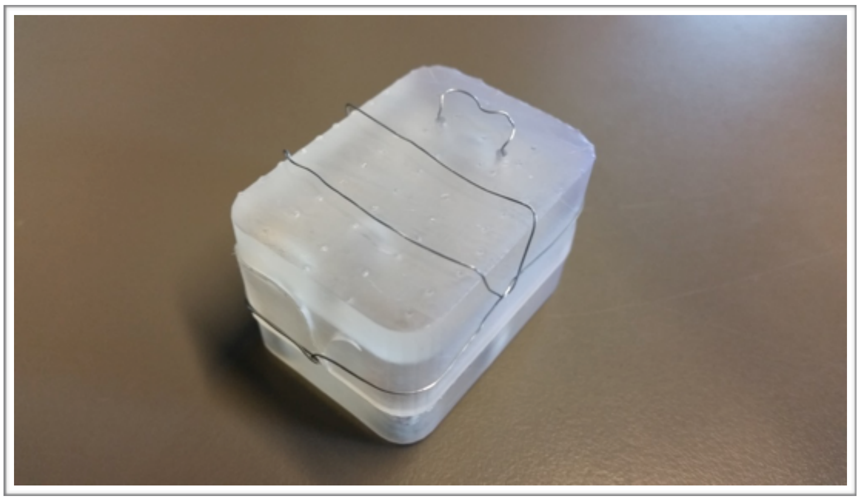


Figure 18: Fourth prototype with top added and wires run around the exterior of the top.

Prototype 5

This prototype had a few notable improvements to the prior prototypes. Here the system using the spring coils as wires to connect the nail touch points and lead user capacitance from the exterior of the widget was introduced and tested (Figure 19). It eliminated the need to attach the wire to the nails removing one challenge, but also allowed the nail themselves to move more freely and without potentially pulling on each other.

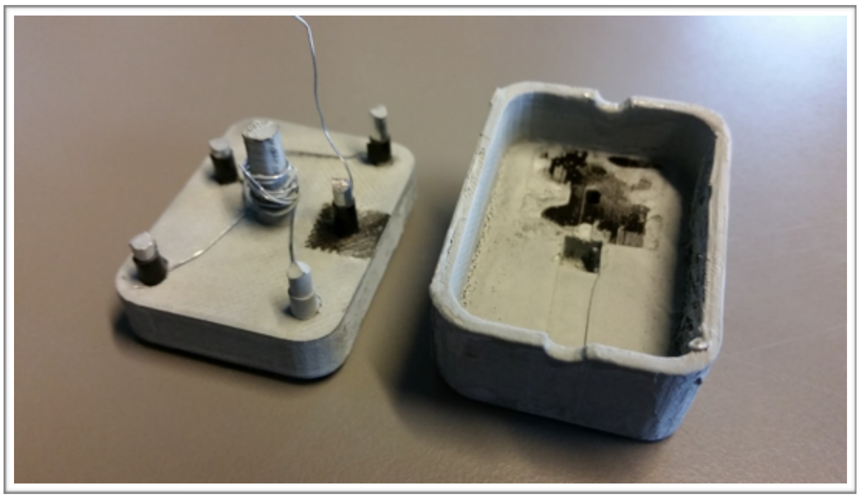


Figure 19: interior of the fifth prototype with spring coils used as wire and zinc paint added.

The button platform was added to make the button stand out more and get closer to the feel of actually pressing a button even if there is no depression or click effect when using it (Figure 20). The raised platform also helped to clear the button wire from the exterior of the widget. It made it easier to make sure the button wire did not interact with the other wires on the exterior that connected with the other touch points and clear up any potential interference from these.

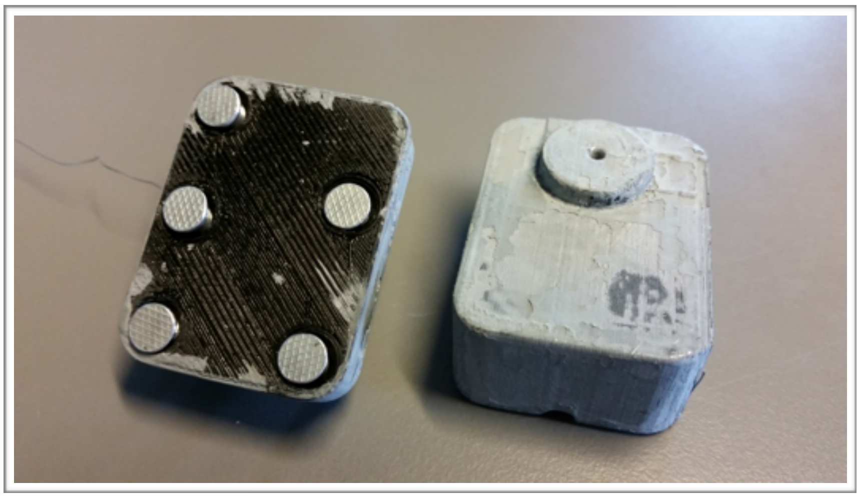


Figure 20: Exterior of the fifth prototype with button platform added and zinc paint applied.

This prototype attempts to maximize the exterior surface area of the widget that could be touched to lead a user's capacitance to the touch points. The first attempt was using a conductive paint, in this case zinc spray-paint, to cover the outside and inside of the widget except for the bottom, the button platform and the interior part with the button wires (Figure 19 and 20). This would allow for almost any shape of the outside of the widget to be used as only a layer of paint need be applied to allow the users capacitance to reach the touch points. It might even be possible to paint on top of it in a thin layer and still have the paint be conductive allowing for further customization of the widget.

The zinc paint did not work that well however only allowing the touch points to be detected by the touch screen sporadically even with several coats applied. Instead metallic tape was used to cover the widget exterior on top of wires leading down to the springs and nails. This enabled the user's capacitance to reach the touch points much better with only minor loss of detection.

4 Widget Detection Algorithm

In order to have the touch device detect the 5-point tangible widgets that we created, an algorithm was implemented in C# using the Unity API for touch points. The full source code can be seen in appendix "B. Source Code for Widget Detection Algorithm".

Unity outputs touch points on Android and iOS devices as a collection of 2D vectors indicating the screen position of each touch point currently touching the screen. The objective of this detection algorithm is to convert this array of touch points to an array of correctly detected widgets. As stated earlier, the tangible widgets have 5 touch points (3 detection points, 1 identification point, and 1 interaction point). The detection points (referred to as the middle-, reference- and discriminant point respectively) will first be used to create a list of widgets on the screen. Then it will go through the remaining touch points on the screen to find the identification, and possibly interaction points of the widgets in the list.

The data pertaining to each widget will be saved in a data structure that we defined. Internally in this algorithm it is known simply as "Widget". The algorithm outputs an array of these Widget objects. Table 1 lists the public variables held in the Widget structure.

Variable	Data Type	Comment
Points	int	The indicies of the widget's detection points in the overall touch array
Confidence	float	The widget's overall confidence value
AngleConfidence	float	The widget's angle-confidence value
LengthConfidence	float	The widget's length-confidence value
DataPoints	List<Vector3>	The position of the dataPoints relative to this widget's own coordinate system
Flags	List<int>	The recognized flags associated with this widget
Position	Vector3	The calculated position of the widget in screen coordinates
Orientation	Vector3	The vector pointing forward from the widget

Table 1: A list of the different variables in the Widget structure.

4.1 Confidence

In order for a pattern of potential detection points to be recognized as a widget, it must satisfy 2 rules. First, the three points must form a right triangle. And second, the two legs in said right triangle must be of equal length. However, since we are unable to account for errors in detecting the positions of touch points, a strict enforcement of these rules would make it very hard to successfully detect widgets. Therefore we introduce the concept of confidence.

The confidence of the Widget is a measure of how sure the system is that the given widget is correctly detected. The confidence value is a floating point number between 0 and 1. A confidence value of 1 means the system is completely certain that the widget is detected correctly, while a value of 0 means that the widget is certainly detected incorrectly.

The confidence is calculated by multiplying the widget's LengthConfidence and AngleConfidence, which are explained below. As both LengthConfidence and AngleConfidence are kept in an interval between 0 and 1, the resulting confidence value will also be kept between 0 and 1.

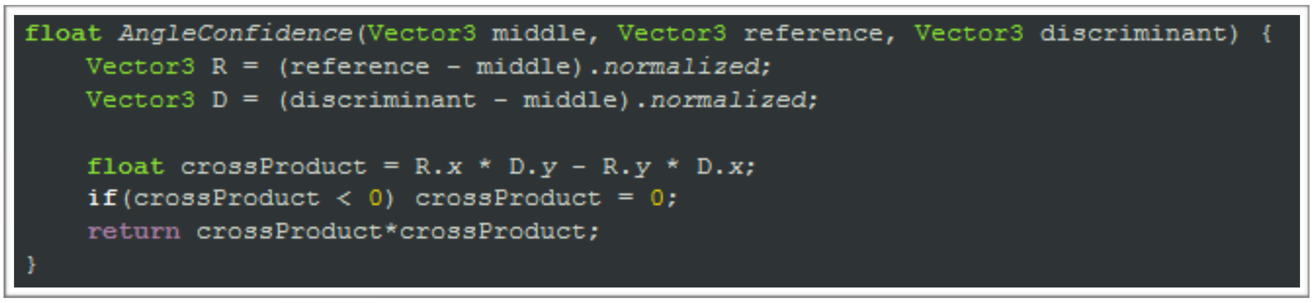
4.2 Angle- and Length Confidence

These two sub-values for confidence, are derived respectively from each of the two rules put forward above. Angle Confidence indicates how closely the widget fit the first rule, with a value of 1 meaning the detection points form a perfect right triangle. Length Confidence indicates how closely the widget fit the second rule, with a value of 1 meaning that the two legs of the right triangle are equal in length.

To calculate the Angle Confidence of three given detection points in a widget, the algorithm first calculates the vector from the middle point to the reference point (R), and then the vector from the middle point to the discriminant point (D). The angle confidence (A) is then calculated using the 2D cross product:

$$A = R_x \cdot D_y - R_y \cdot D_x$$

It is referred to in the code as crossProduct. If the crossProduct is less than 0, it indicates that the discriminant point is on the wrong side of the R vector, which is not allowed for a widget. Therefore crossProduct is clamped to be in the interval [0, 1]. The code calculating the Angle Confidence can be seen in Code Snippet 1.



```
float AngleConfidence(Vector3 middle, Vector3 reference, Vector3 discriminant) {
    Vector3 R = (reference - middle).normalized;
    Vector3 D = (discriminant - middle).normalized;

    float crossProduct = R.x * D.y - R.y * D.x;
    if(crossProduct < 0) crossProduct = 0;
    return crossProduct*crossProduct;
}
```

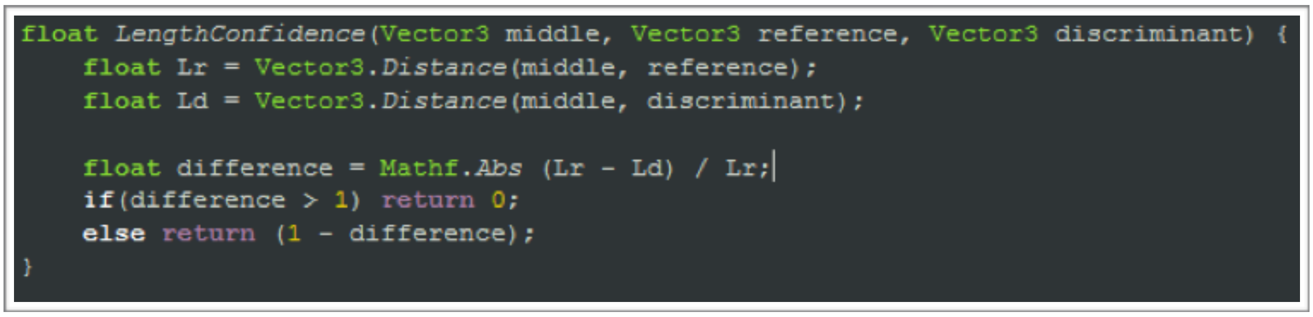
Code Snippet 1: The code for calculating AngleConfidence of three given positions.

Experimentally it was discovered that the calculated Angle Confidence was more reliable if crossProduct is squared.

To calculate the Length Confidence of three given touch points in a widget, the algorithm first calculates the length from the middle point to the reference point (L_r), and the length from the Middle point to the discriminant point (L_d). With these values the length confidence (H) is calculated:

$$H = 1 - \frac{|L_r - L_d|}{L_r}$$

In the event that the nominator is larger than L_r, meaning that the difference between the two lengths is larger than the length of the reference, H could evaluate to a negative number. Therefore H is clamped to be in the interval [0, 1]. A version of the code calculating the length confidence can be seen in Code Snippet 2.



```
float LengthConfidence(Vector3 middle, Vector3 reference, Vector3 discriminant) {
    float Lr = Vector3.Distance(middle, reference);
    float Ld = Vector3.Distance(middle, discriminant);

    float difference = Mathf.Abs (Lr - Ld) / Lr;
    if(difference > 1) return 0;
    else return (1 - difference);
}
```

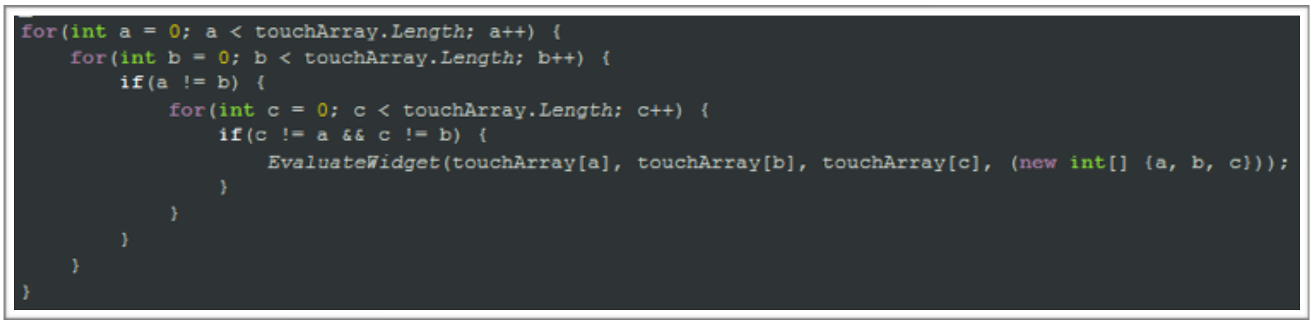
Code Snippet 2: The code for calculating Length Confidence.

With these two numbers, Angle Confidence and Length Confidence, the overall confidence (C) can be calculated by multiplying the two numbers with each other:

$$C = A \cdot H$$

4.3 Detecting Widgets

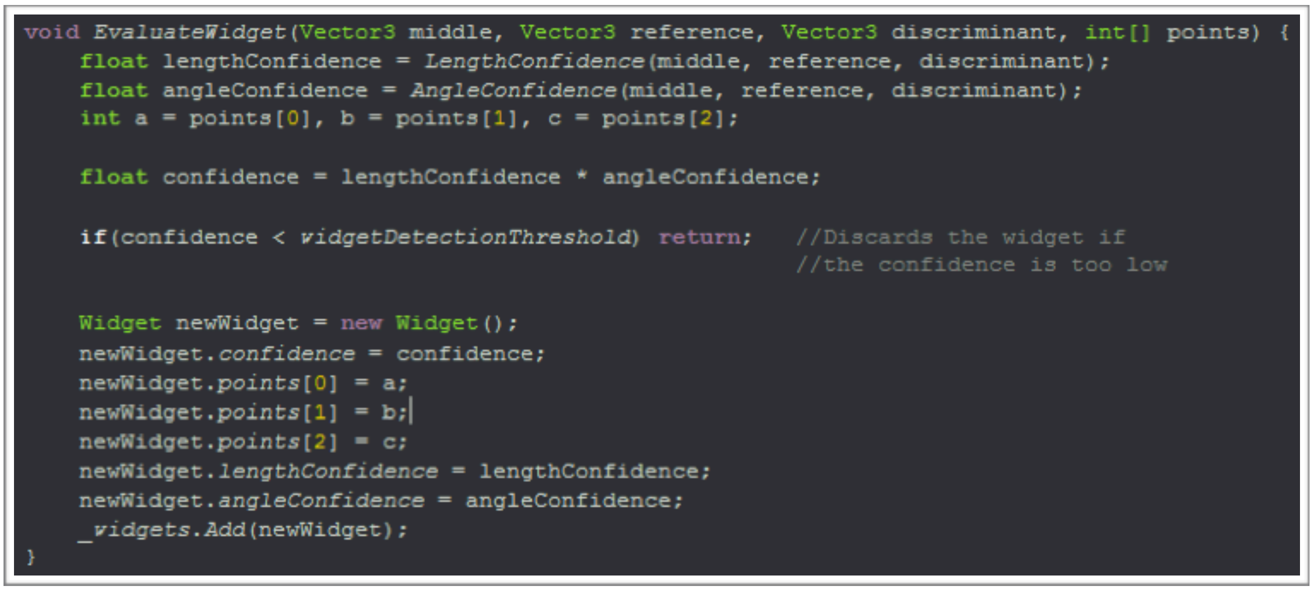
The algorithm calculates the widgets for every frame to be able to detect changes in the widgets. To calculate the widgets, the algorithm calculates confidences for every possible set of three unique touch points. The order of touch points in the set matters in the calculation. Thus, if we say that 0, 1 and 2 are touch points, the algorithm would calculate a confidence value for both the set (0,1,2) and the set (0,2,1). The reason why the order of the points matter, is that the order indicate which detection point the algorithm should assume that the points represent. It assumes the touch points in the set to be (middle, reference, discriminant). Code Snippet 3 shows how the algorithm goes through these sets.



```
for(int a = 0; a < touchArray.Length; a++) {
    for(int b = 0; b < touchArray.Length; b++) {
        if(a != b) {
            for(int c = 0; c < touchArray.Length; c++) {
                if(c != a && c != b) {
                    EvaluateWidget(touchArray[a], touchArray[b], touchArray[c], (new int[] {a, b, c}));
                }
            }
        }
    }
}
```

Code Snippet 3: The code for going through every possible set of touch points. The touchArray field refers to an array of the touch points. EvaluateWidget calculates the confidence value, and ultimately saves the Widget if it is correctly detected

The function EvaluateWidget then calculates the confidence value for the given set, and either saves or discards the Widget. To decide if the widget will be saved or discarded, a threshold is set at compile-time. This threshold indicates how large the confidence value the Widget must have in order to not be discarded. This value can be anything between 0 and 1. In this specific project the value was set to 0.9, chosen through experimentation. Code Snippet 4 shows the code of EvaluateWidget.

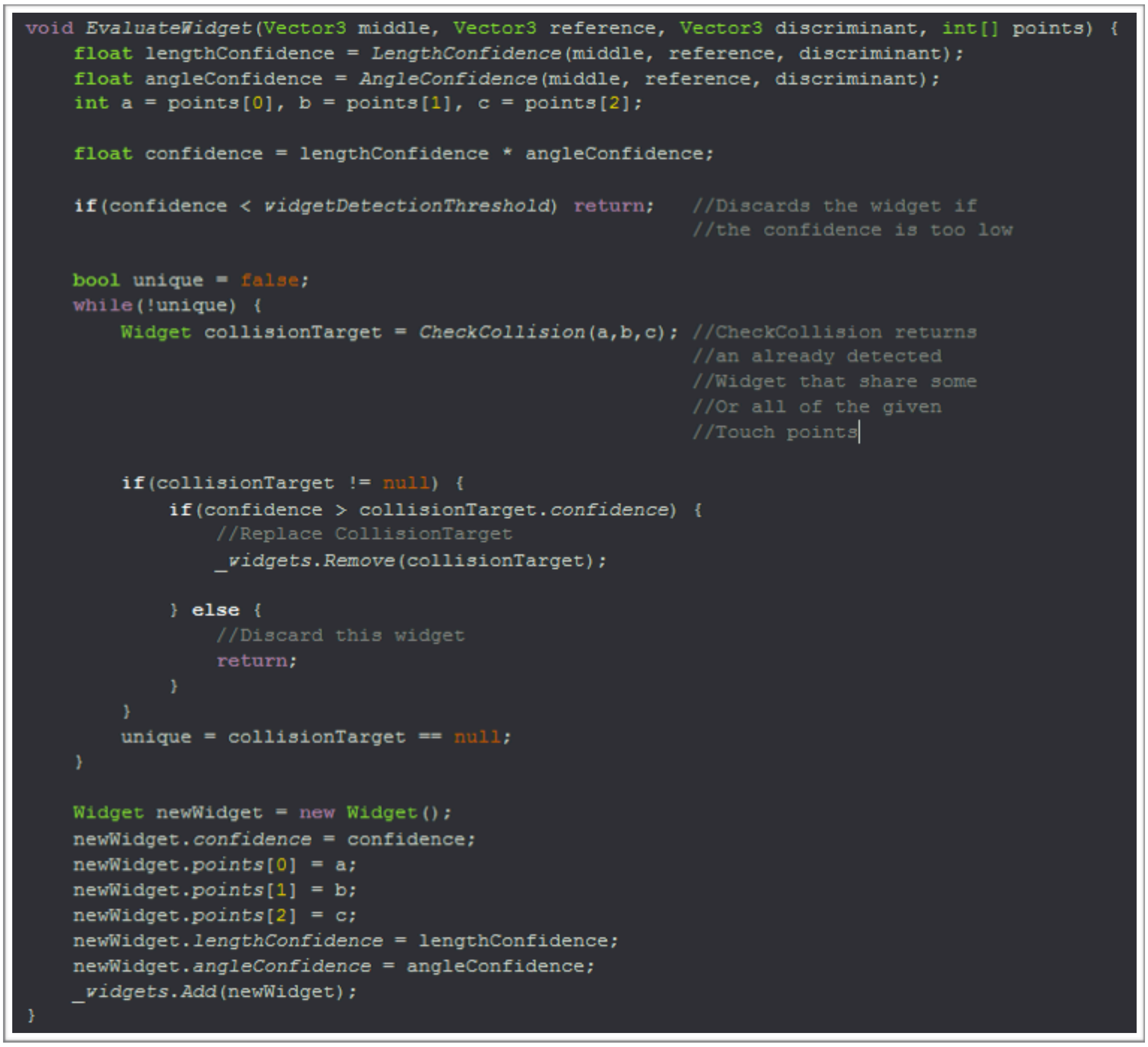


```
void EvaluateWidget(Vector3 middle, Vector3 reference, Vector3 discriminant, int[] points) {
    float lengthConfidence = LengthConfidence(middle, reference, discriminant);
    float angleConfidence = AngleConfidence(middle, reference, discriminant);
    int a = points[0], b = points[1], c = points[2];

    float confidence = lengthConfidence * angleConfidence;

    if(confidence < widgetDetectionThreshold) return;   //Discards the widget if
                                                        //the confidence is too low

    Widget newWidget = new Widget();
    newWidget.confidence = confidence;
    newWidget.points[0] = a;
    newWidget.points[1] = b;
    newWidget.points[2] = c;
    newWidget.lengthConfidence = lengthConfidence;
    newWidget.angleConfidence = angleConfidence;
    _widgets.Add(newWidget);
}
```

Code Snippet 4: The EvaluateWidget function. widgetDetectionThreshold is the given threshold value _widgets is the array where the detected widgets are held.

4.4 Collision Checking

There is an issue with the version of the code in Code Snippet 4. Namely that it allows for detecting widgets that share touch points. Widgets cannot share touch points as every touch point is attached to just one widget, and allowing widgets in the algorithm to share touch points would lead to unpredictable and wrong results. Therefore the EvaluateWidget function needs a step between checking for widgetDetectionThreshold and assigning the new widget. It goes through the already detected widgets to see if the new widget shares touch points with the old one. If it does, it will compare the confidence values of the two. If the old widget's value is larger, it will discard the new widget, and stop execution of the EvaluateWidgets function. If the new widget's value is larger, the old widget will be removed from the array. We refer to this part of the code as "Collision Checking" as it detects collision between the touch points of the widgets. This code makes sure the algorithm only detect widgets that have unique touch points. The augmented version of the EvaluateWidget function can be seen in Code Snippet 5.



```
void EvaluateWidget(Vector3 middle, Vector3 reference, Vector3 discriminant, int[] points) {
    float lengthConfidence = LengthConfidence(middle, reference, discriminant);
    float angleConfidence = AngleConfidence(middle, reference, discriminant);
    int a = points[0], b = points[1], c = points[2];

    float confidence = lengthConfidence * angleConfidence;

    if(confidence < widgetDetectionThreshold) return;   //Discards the widget if
                                                         //the confidence is too low

    bool unique = false;
    while(!unique) {
        Widget collisionTarget = CheckCollision(a,b,c); //CheckCollision returns
                                                         //an already detected
                                                         //Widget that share some
                                                         //Or all of the given
                                                         //Touch points

        if(collisionTarget != null) {
            if(confidence > collisionTarget.confidence) {
                //Replace CollisionTarget
                _widgets.Remove(collisionTarget);

            } else {
                //Discard this widget
                return;
            }
        }
        unique = collisionTarget == null;
    }

    Widget newWidget = new Widget();
    newWidget.confidence = confidence;
    newWidget.points[0] = a;
    newWidget.points[1] = b;
    newWidget.points[2] = c;
    newWidget.lengthConfidence = lengthConfidence;
    newWidget.angleConfidence = angleConfidence;
    _widgets.Add(newWidget);
}
```

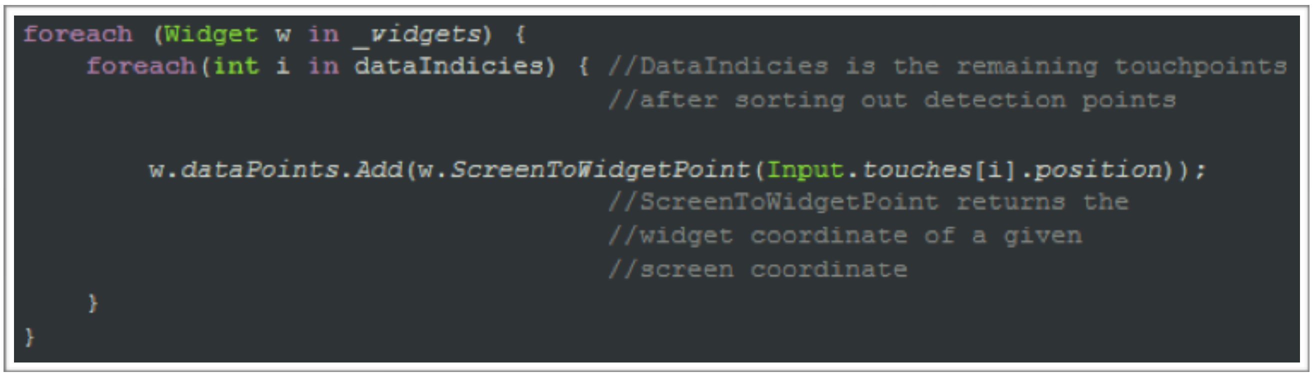
Code Snippet 5: The augmented version of the EvaluateWidget function seen in Code Snippet 4. Added is the code for Collision Checking.

4.5 Data Points and the Widget Coordinate System

When all sets of touch points have been checked for being a widget, it can be assumed that the widgets that have been saved all have high confidence values and unique touch points. So far the algorithm has only been concerned with the three detection points of the widget. With the widgets detected, the algorithm can go on to analyse the remaining touch points for whether or not they are identification points or interaction points. The algorithm does not differ between these two types of touch points, they are analysed in the same manner. It depends on the application to interpret whether or not they are identification points or interaction points. In the algorithm, we collectively refer to these types of points as data points.

The data points can be described as being in a position relative to the detection points. Therefore it is not a stretch to talk about the data points to be positions in a "Widget Coordinate System", which has the reference-vector as the x-axis and the discriminant-vector as the y-axis. The touch points however are given in screen coordinates. Therefore, we need a method of converting these points from screen coordinates to widget coordinates. For this we use matrix manipulation. The conversion will need to alter the position, then the rotation, and then the scale of the given screen coordinate, relative to the position, rotation and size that can be derived from the widget.

For each data point (all the touch points that have not been used as detection points) and for each widget, the algorithm will save the given widget coordinate of that touch point in the Widget's DataPoints variable. Code Snippet 6 shows the code behind that.

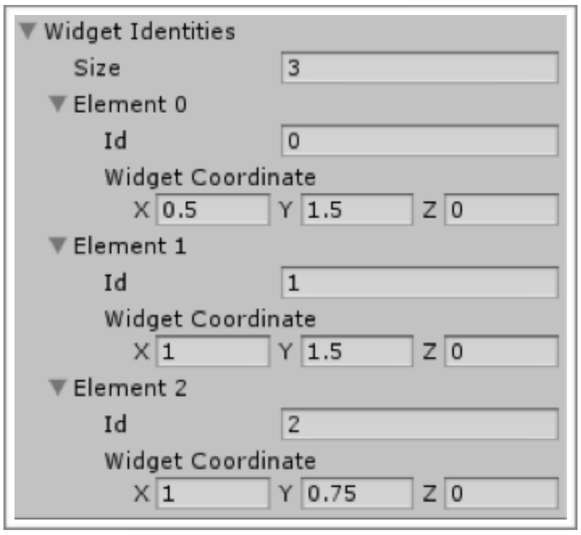


```
foreach (Widget w in _widgets) {
    foreach(int i in dataIndicies) { //DataIndicies is the remaining touchpoints
                                     //after sorting out detection points

        w.dataPoints.Add(w.ScreenToWidgetPoint(Input.touches[i].position));
                                     //ScreenToWidgetPoint returns the
                                     //widget coordinate of a given
                                     //screen coordinate
    }
}
```

Code Snippet 6: The code for adding data points to each widget.

4.6 Widget Flags

Depending on the application, the data points might be utilized in different ways. We found it necessary to add a functionality to register specific widget coordinates to have specific meanings (e.g. (0.5, 0.5) equals a button press). The "Widget Flags" system was implemented to allow the developer to enter "Widget Identities" at compile-time, that can be used to classify widgets at runtime (Figure 21). A widget identity consists of a widget position, and an integer as Id. Briefly said, if a widget has a datapoint at the given widget position, it will also have a flag with the given integer.



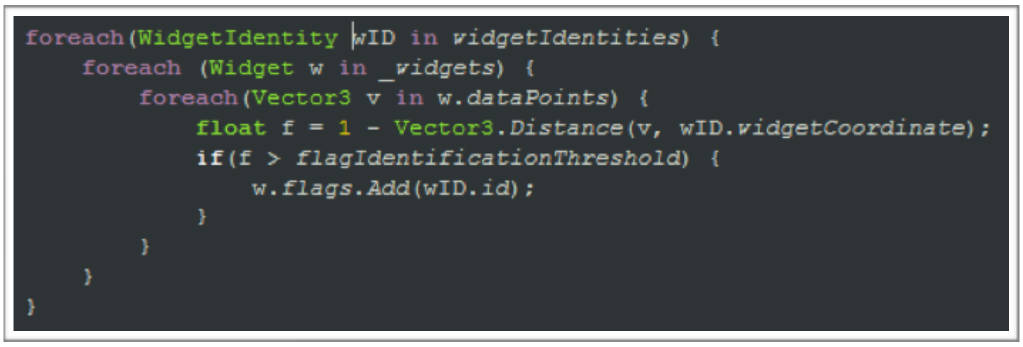
```
▼ Widget Identities
    Size                3
  ▼ Element 0
      Id                0
      Widget Coordinate
        X 0.5    Y 1.5    Z 0
  ▼ Element 1
      Id                1
      Widget Coordinate
        X 1      Y 1.5    Z 0
  ▼ Element 2
      Id                2
      Widget Coordinate
        X 1      Y 0.75   Z 0
```

Figure 21: The interface in Unity where Widget Identities can be input. Currently it holds three different widget identities.

To account for errors in detection, a different Flag-Confidence value (F) is calculated for every datapoint in a widget. The Flag-Confidence is simply:

$$F = 1 - | P_{id} - P_d |$$

where P_{id} is the registered Widget Identity's widget position, and P_d is the given datapoint. If F is larger than a set threshold value, then the widget can be flagged as having the given Widget Identity. The code can be seen in Code Snippet 7.



```
foreach(WidgetIdentity wID in widgetIdentities) {
    foreach (Widget w in _widgets) {
        foreach(Vector3 v in w.dataPoints) {
            float f = 1 - Vector3.Distance(v, wID.widgetCoordinate);
            if(f > flagIdentificationThreshold) {
                w.flags.Add(wID.id);
            }
        }
    }
}
```

Code Snippet 7: The code for adding flags to widgets.

flagIdentificationThreshold is set at compile-time like, but is a separate value from, the widget detection threshold explained earlier.

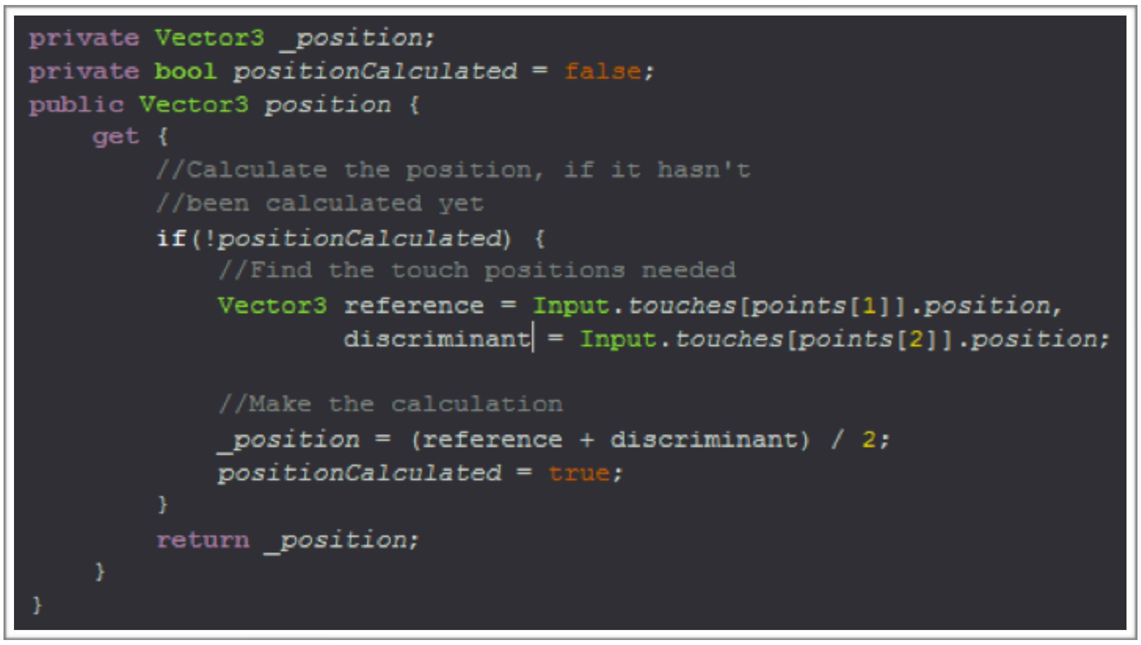
4.7 Position and Orientation

With widgets identified, it is possible to calculate the position and the orientation of these widgets on the screen. Position and orientation are not calculated as part of the algorithm that runs every frame, but rather if and when they need to be accessed from outside the algorithm.

Position (P_w) for the widget was defined as being the midpoint between the reference point (P_r) and the discriminant point (P_d), as this is the midpoint of the widget. It can be calculated as:

$$P_w = (\mathrm{P}_r + P_d) / 2$$

The code for getting the position of a Widget object can be seen in Code Snippet 8.



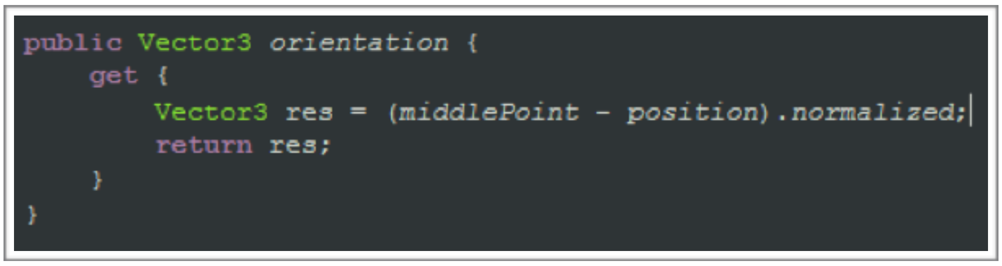
```
private Vector3 _position;
private bool positionCalculated = false;
public Vector3 position {
    get {
        //Calculate the position, if it hasn't
        //been calculated yet
        if(!positionCalculated) {
            //Find the touch positions needed
            Vector3 reference = Input.touches[points[1]].position,
                    discriminant = Input.touches[points[2]].position;

            //Make the calculation
            _position = (reference + discriminant) / 2;
            positionCalculated = true;
        }
        return _position;
    }
}
```

Code Snippet 8: The code for getting the position. Note that the calculation is only made the first time the field is accessed (when positionCalculated is false).

Orientation, or the widget's forward vector (V_f), is defined as the normalized vector from the widget's midpoint to the widget's forward point. For midpoint we can use the position variable (P_w) from before, and for the widget's forward point, the middle touch point (P_m) of the widget is used. The calculation could look like this:

$$V_f = \frac{P_m - P_w}{|P_m - P_w|}$$

The code can be seen in Code Snippet 9.

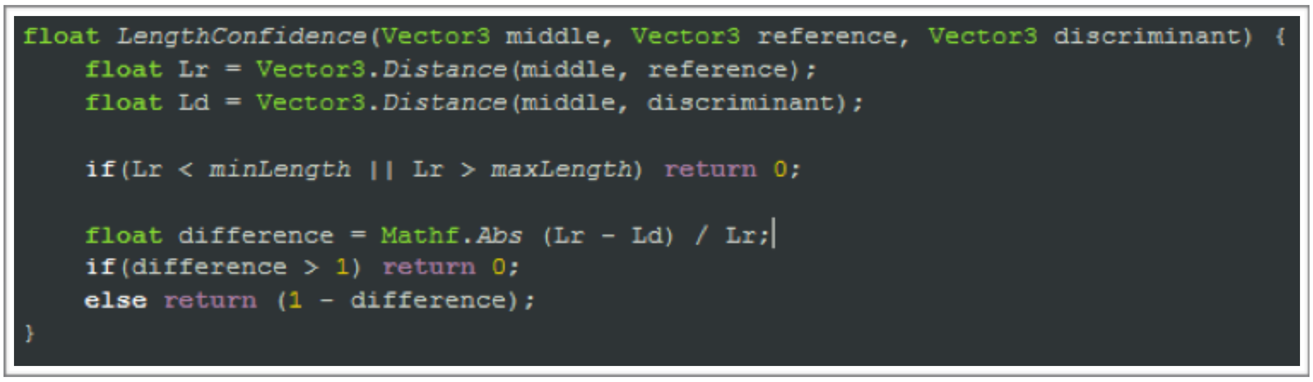


```
public Vector3 orientation {
    get {
        Vector3 res = (middlePoint - position).normalized;
        return res;
    }
}
```

Code Snippet 9: The function for getting the orientation of a widget.

4.8 Corrections to the Algorithm

The finished algorithm did not always work as expected. Therefore it was necessary to make corrections, in order to make it work as intended. For example, the algorithm showed to have trouble detecting a widget when there were additional touch points on the screen. This called for a way to sort out touch points that were too far away from the widget. This was implemented by adding some code to the LengthConfidence function (which can be seen in Code Snippet 2). Code Snippet 10 shows the improved version of the LengthConfidence function.



```
float LengthConfidence(Vector3 middle, Vector3 reference, Vector3 discriminant) {
    float Lr = Vector3.Distance(middle, reference);
    float Ld = Vector3.Distance(middle, discriminant);

    if(Lr < minLength || Lr > maxLength) return 0;

    float difference = Mathf.Abs (Lr - Ld) / Lr;
    if(difference > 1) return 0;
    else return (1 - difference);
}
```

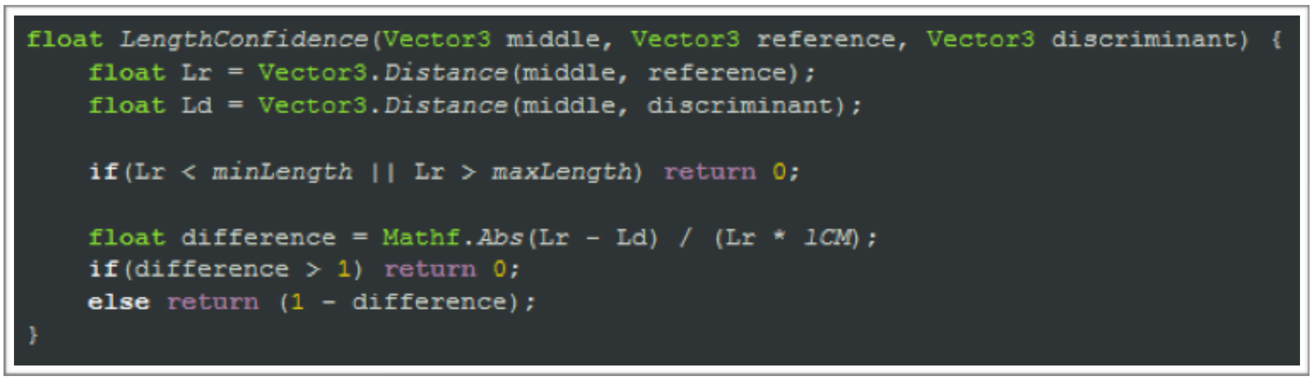
Code Snippet 10: The altered version of the LengthConfidence function. The added line will return a 0 confidence, if the reference length lies beyond the interval defined by minLength and maxLength.

The two variables, minLength and maxLength, are values given in absolute screen coordinates. They are set prior to use by a separate program that can measure the length in screen coordinates between touch points of a widget. The measured length ($L_{measured}$) can then be used to calculate the minLength (L_{min}) and maxLength(L_{max}):

$$L_{\min} = L_{measured} \cdot 0.9$$
$$L_{\max} = L_{measured} \cdot 1.1$$

Thus, only touch points that have the correct distance to one another, can be used as detection points in the algorithm.

The detection of the widgets also became unreliable when we tried to make the overall size of the widgets smaller. Through experimentation it was found that as the decrease in size had no influence on what AngleConfidence was calculated to be, it did have an influence on the calculation of LengthConfidence, making it more prone to error. Therefore, it was necessary to correct the calculation of LengthConfidence further. Code Snippet 11 shows how this was done:



```
float LengthConfidence(Vector3 middle, Vector3 reference, Vector3 discriminant) {
    float Lr = Vector3.Distance(middle, reference);
    float Ld = Vector3.Distance(middle, discriminant);

    if(Lr < minLength || Lr > maxLength) return 0;

    float difference = Mathf.Abs(Lr - Ld) / (Lr * lCM);
    if(difference > 1) return 0;
    else return (1 - difference);
}
```

Code Snippet 11: The final version of the LengthConfidence function.

lCM in the code stands for lengthCorrectionMultiplier. It is a value that is set by the developer at compile-time. Experimentally it was found for this project that a lCM of 2 helped combat the problems related to widget size.

4.9 Result

This chapter has described the considerations and implementations behind the detection algorithm used in this project. With this code it is possible, at all times, to get information about how many widgets are on the screen, where they are, how they are oriented, and identify their data points. This allows for use of widgets in a tablet game. No assumptions are made in the development of the widgets about what kind of game design the widgets are to be used for, and therefore the algorithm should be able to be used in multiple different game designs.

5 Game Design

This section will explain the different design aspects of the game Hover Wars, which was created with the purpose of testing the hypothesis of this Project project. The game design is based on theories and guidelines explained in various books about game design (Rogers, 2010; Fullerton, 2014; Schell, 2014).

5.1 Core Gameplay

Hover Wars is a fast-paced game made for iPad, where two players will have to compete against each other in a “kill or be killed” battle on the same screen. In the game, each player has control of their own hover tank, which has the capabilities of moving, rotating, shooting projectiles, hovering (to become invulnerable) and picking up power-ups. Each tank has a certain amount of energy, which will be depleted in various ways, such as when shooting, hovering or taking damage. When the energy reaches 0, the hover tank will explode and the player controlling it has lost the game. Besides controlling the hover tank, different power-ups will appear at random intervals and locations on the battlefield. These power-ups work as an aid to the player who collects them. The goal of the game is to attack your opponent by shooting and picking up power-ups while trying to dodge your opponents attacks at the same time. The game can be seen in Figure 22.

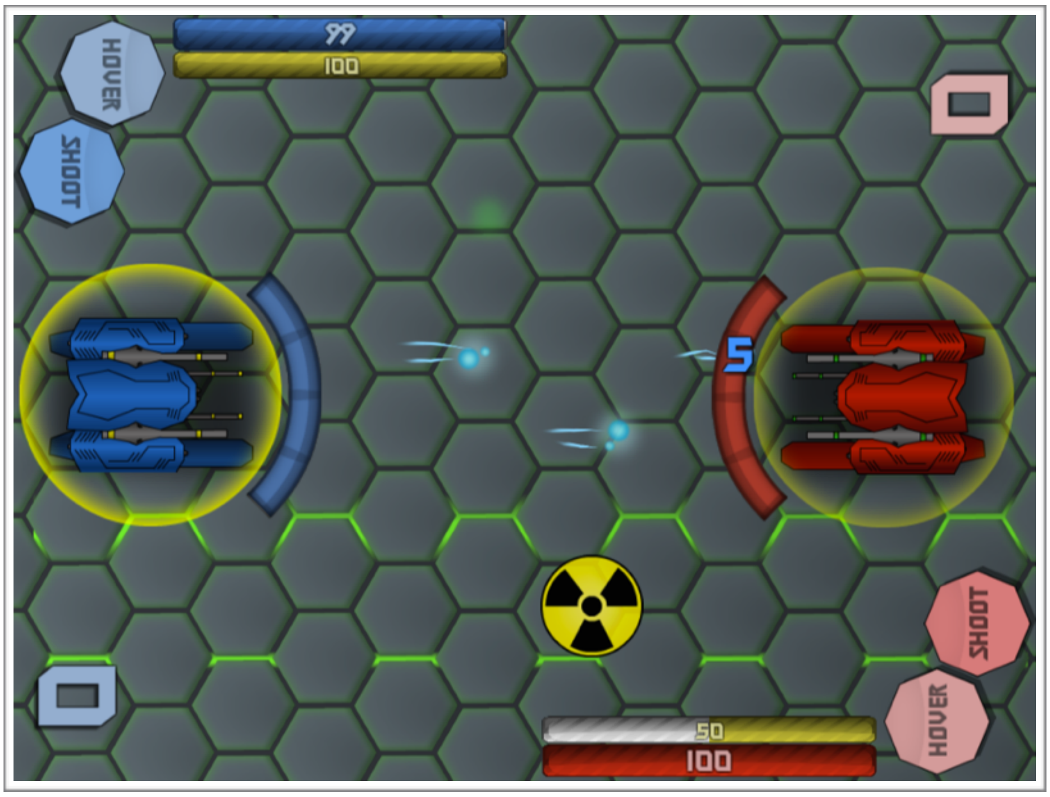


Figure 22: Overview of what Hover Wars looks like during a game session. The blue player shoots some projectiles towards the red player. Near the bottom of the screen a “Nuclear Blast” power-up is available to be collected by either of the players.

5.2 Game Design Around Widgets

The functionality of the hover tank and the gameplay in Hover Wars, was designed around five key features, which is afforded by the design of the widgets. This means that when designing the game, the control scheme with touch interaction has not been taken into consideration, until the game design was done. Afterwards, through multiple tests (see section 6.1 Interaction Tests), an optimal control layout was designed for interacting with the game through touch.

The five key features which is afforded by the use of a widget are as follows:

- Movement
- Rotation
- Identification
- A physical button
- Detection (presence on the screen)

When designing the game, the goal was to make it possible to play the game entirely by using the widget, without the need of any GUI. To make sure movement, rotation and identification would become a key feature in the game, it was decided to make a competitive game, where two players (identification) would have to attack and avoid each other. Therefore we created an open battlefield where both players could move and rotate freely, providing the possibility of out manoeuvring the opponent (e.g. moving next to the opponent and shoot him from the side). By having a game where it was possible to shoot at each other, it was decided to use the physical button on the widget as a shoot button. The final key feature of the widgets that we needed to create a game feature for was the detection. At the same time the game needed a game mechanic which made it possible for the players to avoid the attacks from the opponent. Therefore we came up with the idea of making it possible for the tanks to hover, meaning they could be "removed" from the game and by that become invulnerable to attacks. This also fitted the idea of being able to detect the widgets when they were present on the screen. So when the widget was positioned on the screen, the hover tank would not hover, and when the widget was lifted from the screen, the hover tank would start hovering.

With this core game design in mind, we looked further into how to make the game interesting using different game mechanics.

5.3 Energy

Energy is the main resource in the game. When designing how the players should be able to lose a game, we decided when reaching 0 energy, the hover tank would explode and the game would be lost. But we thought that we could do more with the energy than just having it functioning as an indication of when the player was about to lose.

We came up with a design where energy would be used for the following things:

- Taking damage
- Hovering / not hovering
- Shooting projectiles

An amount of energy would be drained whenever the hover tank got hit by any of the opponents attacks. To make it possible to be strategic about the remaining energy, we made it possible for the players to recharge the energy over time. This was done by being present on the screen (not hovering). This makes it possible to go back to full energy, if the player can avoid enough attacks from the opponent over time some time. To make sure the game would be fast-paced, we decided that hovering would drain energy over time. This would prevent the players from hovering all the time, but have them to interact with the game instead. The player would then have to decide if it was best to take a small amount of damage from the opponents attacks or if it would be more preferable to hover for a short moment to avoid the attacks. The final thing that could affect the energy was when the player was shooting. For every projectile that was shot a small amount of energy was drained. In the game it is possible to “auto fire” by holding down the shoot button or to “rapid fire” by tapping the shoot button faster than the firing rate of the auto fire. The amount of energy that is drained per shot is calibrated so when auto firing, it takes up the same amount of energy that is being recharged for no hovering. This means that auto fire negates the energy recharged, and if the player chooses to rapid fire, some energy will be lost over time.

Besides having energy as the only resource, a shield mechanic was added (Figure 23). When the hover tank is taking some damage, the damage goes to the shield before it begins to be drained from the energy. This means that the shield has to be destroyed before it is possible to do damage to the energy, and by winning the game.

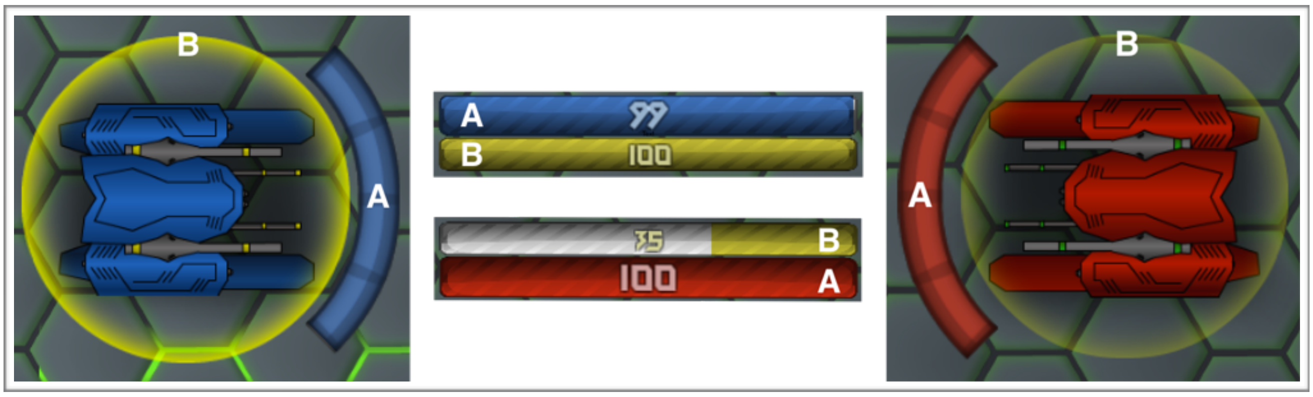


Figure 23: The in-game graphics which indicates how much energy and shield each of the players has left. (A) Shows the remaining energy for both players. (B) Shows the remaining shield for both players. The red player only got 35 shield left, which is shown both by the number inside the shield bar but also by the surrounding shield of the red hover tank becoming more transparent.

5.4 Power-ups

To make the game more varied, we added several power-ups. A power-up in Hover Wars is an item that spawns at a random position on the battle-field. There would spawn a new power-up with a random time interval (3 to 10 seconds). To collect a power-up one of the players would have to move his hover tank over it. Before a power-up is spawning, a short animation would appear in the position where the power-up would come (Figure 24). The reason for having a small animation to begin with, is to make the players aware of a new power-up would spawn.

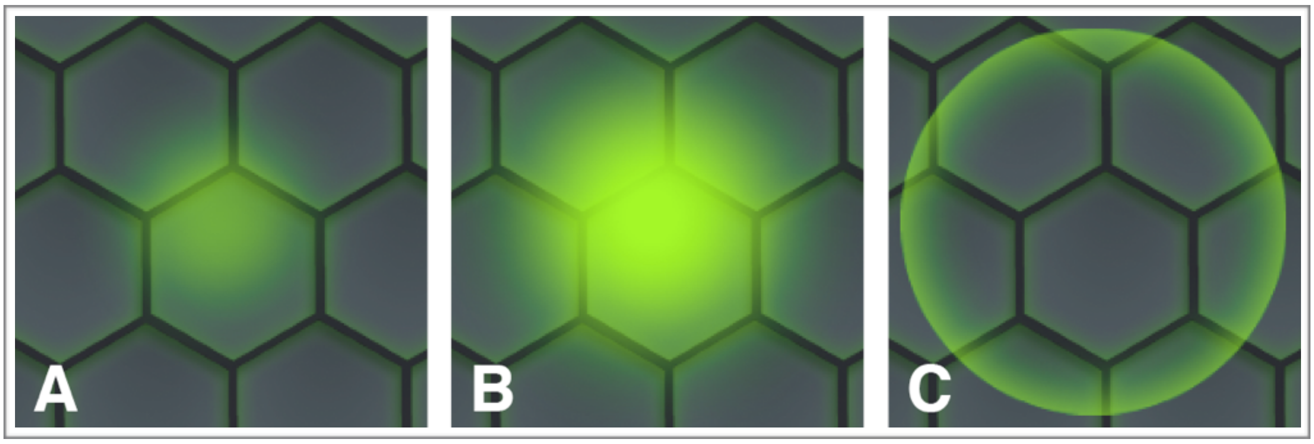


Figure 24: An animation that is played before a power-up is spawned on the battlefield. The animation pulsates three time between a small glow (A) and a large glow (B) whereafter a small explosion ring appears and increases in size (C).

There have been a huge variety of possible power-ups which we wanted to implement into the game, but we restricted ourselves to only implement 5 power-ups in total. The reason for this was that we wanted to keep the goal simple. If we implemented too many power-ups, then it would take too long time for new players to learn the game and the behaviour of the different power-ups. The design goal of the chosen power-ups has been to benefit the player who picks them up, in contrast to weakening the opponent.

The power-ups which made it to the game are as follows:

- Spider Robots
- Laser Turret
- Nuclear Blast
- Reflector Shield
- Recharge Shield

Besides adding power-ups to the game, we also discussed adding some hazards to the game. Hazards were meant to be the opposite of a power-up. Whenever one of the players picks it up, that player to be punished by either losing some energy, be stunned, or get the weapons jammed. Hazards would have the same spawn animation in the game as the power-ups, meaning it would not be possible to differ between power-ups and hazards until the icon would show up. The idea behind adding hazards to the game was to make sure the players would not stay on top of a power-up/hazard spawn animation, but would wait until it spawned and be strategic about which ones should be picked up or not. In the end we decided not to go with the hazards. The reason for this was that the main design goal of the game was to award each of the players for interacting with the game instead and not punishing them, which the hazards would do.

One more feature that was discussed, but not added to the game, was weapon upgrades. The weapon upgrades were meant to permanently increase the damage of the projectiles, meaning it would make it easier to kill the opponent. There were multiple reasons to why it was decided not to add the weapon upgrades. First of all, the weapon upgrades could give an imbalance in the game, making one of the players more powerful than the other. Besides that, the goal was to have a fast-paced game. If weapon upgrades should make any sense, each game session would have to be longer (a game takes about 17 seconds on average). The final reason was we wanted to keep the features in the game to a minimum so it would be fast and easy to learn the game.

Spider Robots

This power-up will spawn 5 small spider robots when it is being picked up. Each of the spider robots will crawl towards the opponent and explode on impact. They have a life time of 3 seconds before they destroy themselves. If the opponent hovers while the spider robots are active, they will be standing still and be waiting for the opponent to come back to the ground. The spiders can not explode when hitting the player who picked up the power-up. The idea behind the spider robots is to make the players move around on the battlefield and use the hover function to avoid them. This is achieved by having the spider robots covering an area of the battlefield which is dangerous. The player has to be in constant motion to avoid the spider robots.

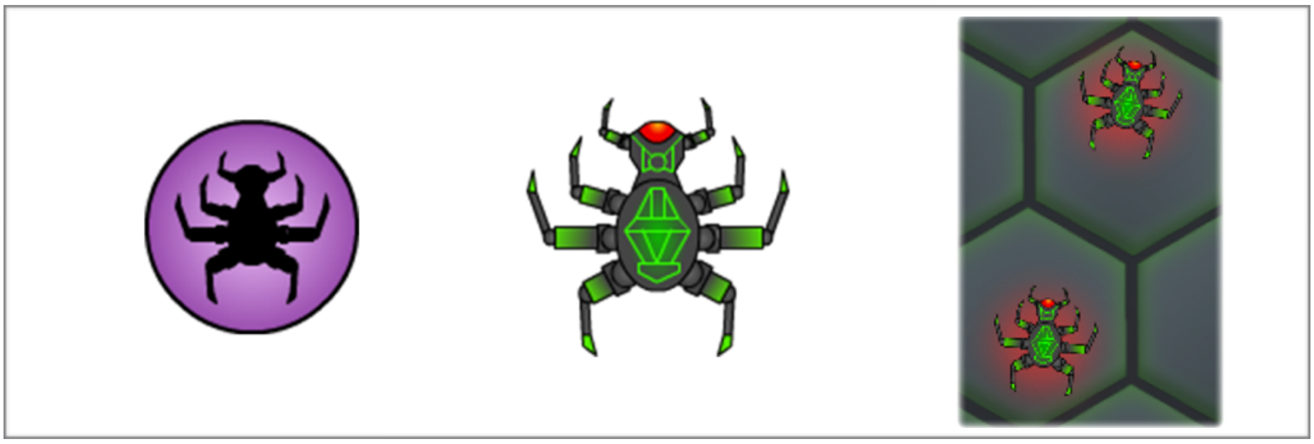


Figure 25: (Left) The icon for the spider robots power-up. (Middle) The visual design of a spider robot. (Right) In-game graphics of two spider robots that have been picked up by the red player. This can be seen by the robot spiders having a red glow underneath them.

Laser Turret

The laser turret is a power-up that deploys a turret on the location where the power-up was picked up. The laser turret can rotate 360 degrees but can not move. It targets the opponent and slowly rotates towards the opponents position. It has a life time of 5 seconds, whereafter it will be destroyed and removed from the game. While the laser turret is active it shoots a constant laser beam which deals damage to the first hover tank it hits. This means it can also hit the player who picked it up if he gets in the way, but it will target and rotate towards the opponent. The idea behind the laser turret is to make the players move around on the battlefield and use the hover function to avoid it. This is achieved by having a moving laser beam that splits the battlefield into two sides. The players has to be in constant motion to avoid the laser beam.

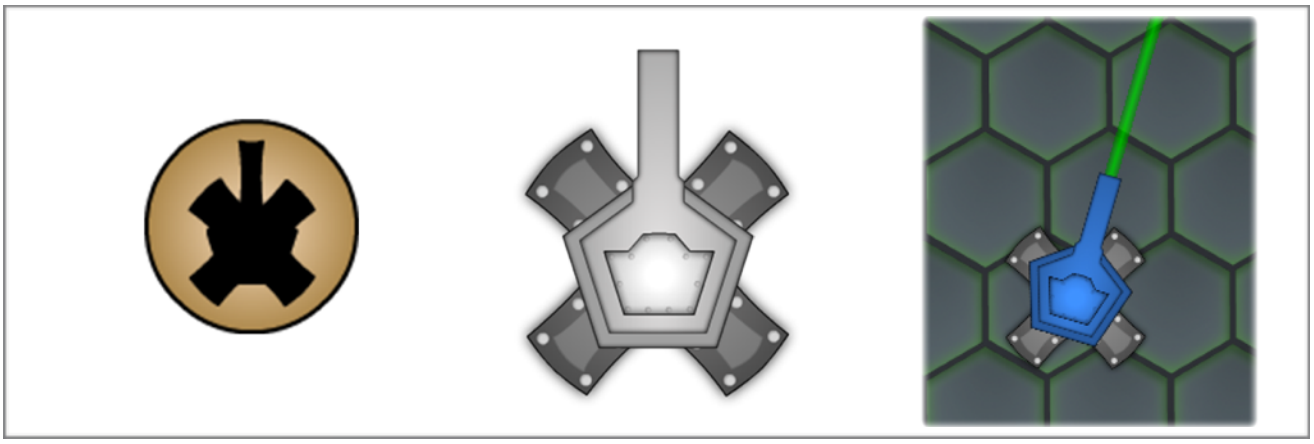


Figure 26: (Left) The icon for the laser turret power-up. (Middle) The visual design of the laser turret. (Right) In-game graphics of a laser turret that has been picked up by the blue player. This can be seen by the color of the turret.

Nuclear Blast

A massive explosion that causes massive damage to all non-hovering tanks. When this power-up is being picked up, a warning sound and animation begins playing for a few seconds. When the warning is over, a nuclear blast occurs and deals damage to both players, unless they are hovering in the moment the explosion takes place. This is a neutral power-up that does not benefit any of the players, except that the one who picks it knows that a nuclear blast warning has been activated, and because of that might be more aware of when he has to hover to avoid the explosion than the opponent. This power-up is designed to give the players an extra incentive to use the hover function of the hover tanks when playing the game.

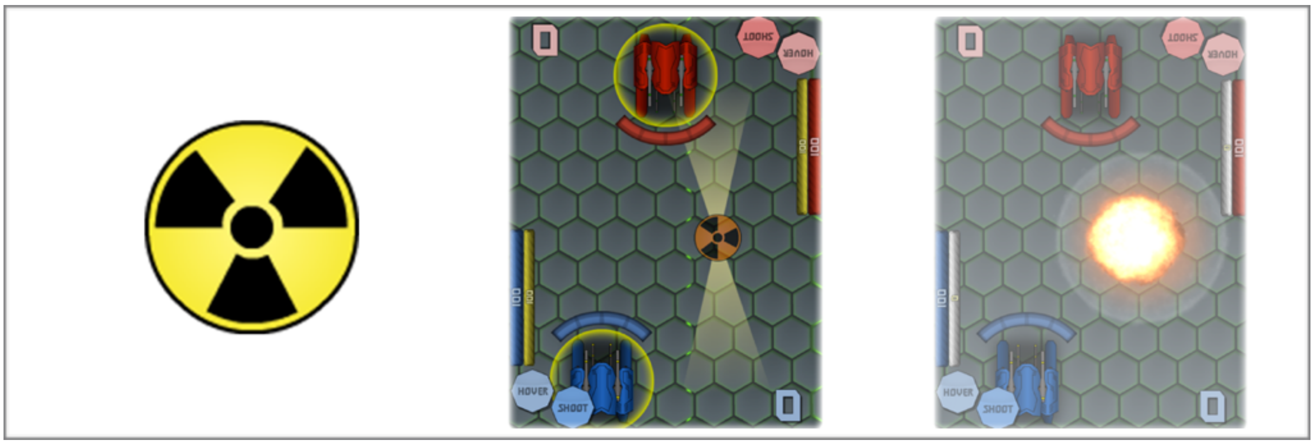


Figure 27: (Left) The icon for the nuclear blast power-up. (Middle) The nuclear blast power-up has been picked up by a player, and the warning animation has begun. (Right) The warning animation is done and a nuclear blast animation is being played. The screen flashes white for a short moment and the explosion can be seen.

Reflector Shield

The reflector shield adds an extra shield around the hover tank for 5 seconds. While the reflector shield is active all of the opponents projectiles and laser beams will be reflected, meaning no damage would be taken and adds the chance of hitting the opponent with his own projectiles when they fly back towards him. The player can still take damage from spider robots and nuclear blasts while having an active reflector shield. The reason for having this power-up is to give the player a feeling of being invulnerable for a short amount of time.

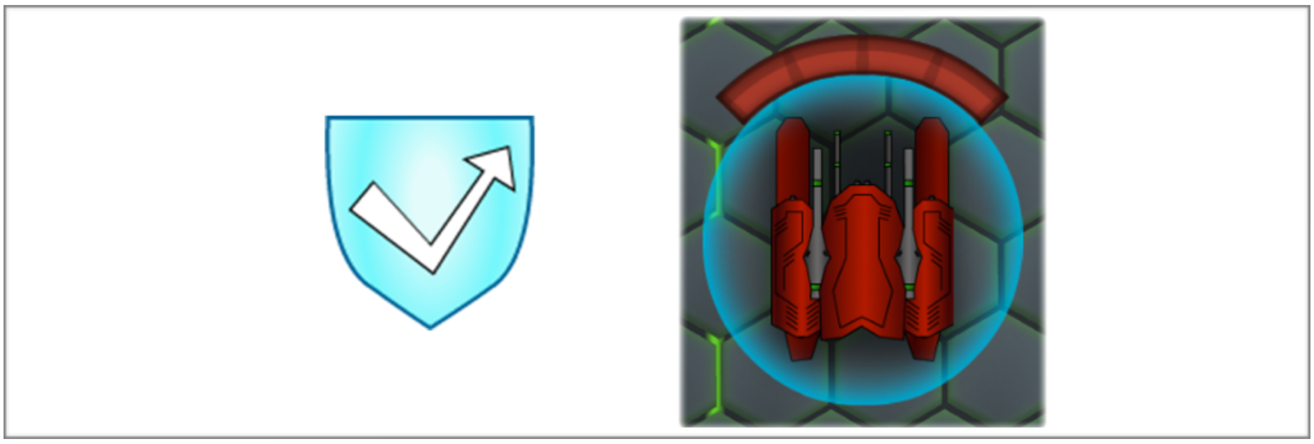


Figure 28: (Left) The icon for the reflector shield power-up. (Right) In-game graphics of what it looks like when a player has the reflector shield activated on his hover tank.

Recharge Shield

The hover tank that picks up this power-up will get its shield fully recharged. This makes sure the hover tank can withstand some more damage from the opponent, making it more difficult to get killed. This power-up was designed to give the chance of getting back into the game if the player was low on energy, has lost his shield and was about to lose the game.

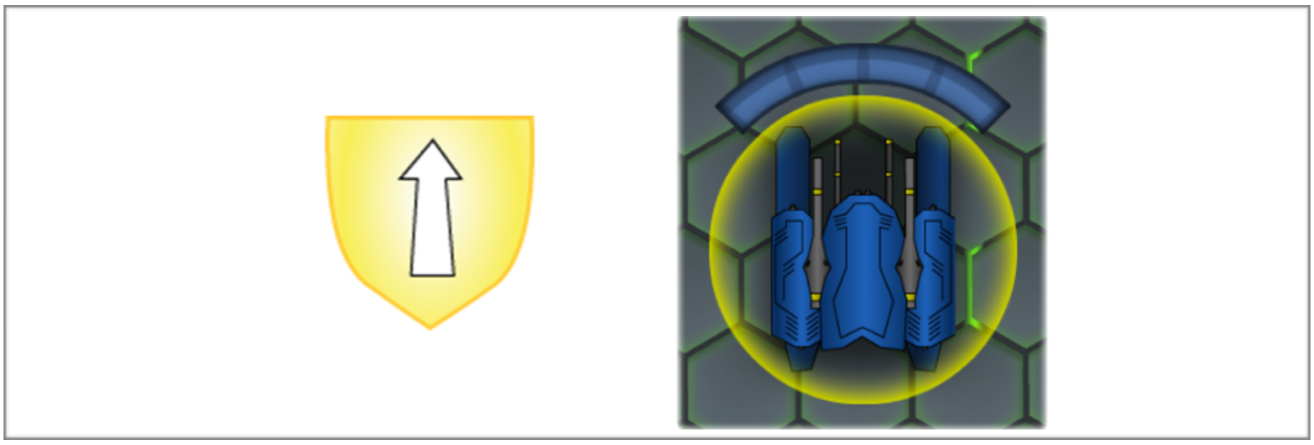


Figure 29: (Left) The icon for the recharge shield power-up. (Right) In-game graphics of what it looks like when a player has a fully charged shield surrounding his hover tank.

5.5 Controlling the Game

To be able to control the hover tank in the game, we have implemented two different methods of interaction: Touch Interaction and Widget Interaction. Based on tests (see section 6.1 Interaction Tests) we came up with the best possible solution for controlling the hover tank, both for touch and widget.
The player has the following options of interaction, when he wants to control the hover tank:

- Movement
- Rotation
- Hovering
- Shooting

In this section the two interaction methods will be explained, covering some of the early control designs and also the final control design which made it to the game.

Touch Controls

Due to the game being designed around the key features of the widgets, it became difficult to come up with the best possible design for controlling the game with touch interaction. Therefore various control designs were implemented and tested.

Movement & Rotation

First off we had to figure out how to move and rotate the hover tank. The first control design had a move icon and a rotate icon (Figure 30) on the hover tank, where the player would have to touch and hold on either of the icon to move and/or rotate the hover tank. In the next iteration we removed the move icon from the hover tank and instead made the entire tank become a “move icon” (Figure 30). This meant that it became possible to touch the entire hover tank in order to move it around, but it was still a necessity to hold on to the rotate icon in order to rotate the hover tank. Finally we also removed the rotate icon from the hover tank. Instead we made movement and rotation of the hover tank rely on how many fingers the player were holding on the tank at the same time. By holding 1 finger on the hover tank it became possible to move it around. By holding 2 or more finger on the hover tank, it was possible to move and rotate it at the same time. By rotating the hand and changing the alignment of the fingers, the hover tank would rotate the same amount as the fingers did.

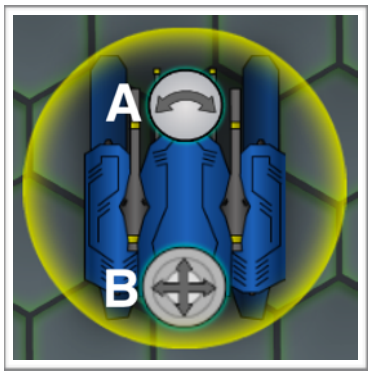


Figure 30: (A) The rotation icon, which had to be touched to rotate the hover tank. (B) The move icon, which had to be held to move the hover tank.

Shooting

The next task was to come up with a good way of shooting projectiles. We tested two different ways of making the hover tank shoot. The first option was to add a GUI button with the label “Shoot” (Figure 31). Whenever the player tapped or held the shoot button, the hover tank would start shooting. The second option was to use the detection of how many fingers the player had on the hover tank at the same time. Since we already used the one and two fingers for moving and rotating the hover tank, we decided that if a third finger was present on the hover tank it would make the hover tank start shooting.

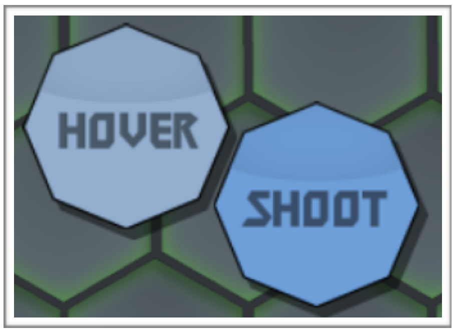


Figure 31: GUI buttons for the blue player. The left GUI button allows the player to use the hovering function of his hover tank. The right GUI button allows the player to shoot projectiles with his hover tank.

Hovering

The final task was to design a way of making the hover tank use the hovering function. Just like shooting, we added a GUI button with the label “Hover” (Figure 31). When the player held a finger on the hover button, the hover tank would be hovering. Whenever the player released the hover button, the hover tank would stop hovering. This required the player to use two hands to play the game. One hand need for hovering and one hand for moving and rotating. Besides having a GUI element for hovering we also tried to implement two different way of hovering by the use of touching the hover tank or not. The first of these designs made use of the amount of fingers present on the hover tank. If no fingers were touching it, then it would be hovering. If one or more fingers were touching the hover tank, then it would stop hovering. This meant that the player would have to remove the fingers from the tank to be able to hover, and by that the player would not be able to move while he was hovering. The second design made it possible to change position while hovering. In this design the players would have to touch the screen close to their own hover tank (and not necessarily on top of the hover tank itself) whereafter the hover tank would snap to the position on the screen where the player was touching and stop hovering. If the player removed the fingers from the screen the hover tank would start hovering again.

The following touch controls, based on multiple tests (see section 6.1 Interaction Tests), made it to the game:

Movement & Rotation

» One finger on the hover tank to move it and two or more fingers to move and rotate it at the same time.

Shooting

» Having a GUI button in the lower corner of the screen.

Hovering

» Having a GUI button in the lower corner of the screen.

Widget Controls

Because the controls of the game were designed around the key features of the widgets (see section 5.2 Game Design Around Widgets), there have not been as many different control designs for how to play the game when using widgets, as when playing with touch interaction.

To activate the functions of the widget, it has to be held flat against the screen, making all of the touch points underneath it touching the iPad.

Movement & Rotation

When positioned on the screen, the hover tank with the same ID as the widget will snap to the position and orientation of the widget. To move and rotate the hover tank, the widget has to be repositioned on the screen either by dragging it around or by lifting it and placing it somewhere else.

Shooting

Two different control designs for shooting were implemented and tested. The first was to use the physical button on the widget. The player had to hold or tap the button to shoot projectiles. The second control design for shooting was to use a GUI element as a shoot button, just like in the touch version (Figure 31).

Hovering

Like shooting, two different control designs for hovering were implemented and tested. In the first design the player had to lift the widget from the screen and then the hover tank hovered. When the widget was positioned back on the screen, the hover tank stopped hovering. The other control design for hovering was to add a GUI element for hovering. Whenever the GUI button was touched the hover tank was hovering, and when released the hover tank stopped hovering (Figure 31).

The following widget controls, some based on multiple tests (see section 6.1 Interaction Tests), made it to the game:

Movement & Rotation

» Position the widget flat against the screen to make the hover tank snap to the position and orientation of the widget.

Shooting

» Tap or hold the physical button on the widget.

Hovering

» Lift the widget from the screen.

6 Experiment

This section describes the various tests that have been made during this project.

6.1 Interaction Tests

Prior to the main preference test, we made five different interaction tests, where we tested six different interaction conditions. In one of the five tests, we tested two conditions at the same time. All of these tests were informal. The purpose of these test were to find the best possible interaction methods for playing Hover Wars, both when using touch interaction and widget interaction. While doing these test, we asked the test participants about what they felt about the game design and if they noticed any bugs in the game. By doing this we could both get other peoples opinion about how the interaction with the game felt and we could find bugs and game design flaws at the same time.

Each interaction test was conducted as a comparative test, where the test participants had to try two different versions of the same game but with two different interaction methods. Four of the tests were about touch interaction and two tests were about widget interaction. All interaction tests were based around a semi-structured interview while the test participants were playing the game.

In the game design section a more in depth description of the different interaction methods are explained (see section 5.5 Controlling the Game).

Touch interaction tests:

- Shoot button in the corner of the screen vs. Shoot button located on the tank **(Touch A)**
- Not touching the tank vs. touching a GUI button to use the hover function **(Touch A)**
- Move & rotate icons vs. amount of fingers on the hover tank to move and rotate **(Touch B)**
- Three fingers on hover tank vs. GUI button to shoot **(Touch C)**

Widget interaction tests:

- Lifting the widget vs. GUI button to hover **(Widget A)**
- Using physical button vs. GUI button to shoot **(Widget B)**

	Touch A	Touch B	Touch C	Widget A	Widget B
Male participants	10	9	1	2	9
Female participants	0	1	3	2	3
Total	**10**	**10**	**4**	**4**	**12**

Table 2: The amount of test participants throughout the five different interaction tests.

All of the test participants were students at Aalborg University from various studies, and none of the test participants have tried more than one of the five different tests.

Based on the results from these interactions tests, multiple changes were added to the game. These changes concerned the interaction design for both touch and widgets, but also concerned features and feedback in the game design, making the game more appealing to the players.

The final touch interaction method became:

Movement & Rotation

» One finger on the hover tank to move it and two or more fingers to move and rotate it at the same time.

Shooting

» Having a GUI button in the lower corner of the screen.

Hovering

» Having a GUI button in the lower corner of the screen.

The final widget interaction method became:

Movement & Rotation

» Position the widget flat against the screen to make the hover tank snap to the position and orientation of the widget.

Shooting

» Tap or hold the physical button on the widget.

Hovering

» Lift the widget from the screen.

Other changes and features that were added to the game based on comments from the test participants:

- Added floating text for visual feedback of how much damage were dealt.
- Color coding bullets, floating text etc. so each player could see what belonged to him.
- Added a permanent energy bar above each hover tank, to make it easier to see the remaining energy.
- Added a win counter, making it possible to see how many times each player had won.

6.2 Preference Test

The main test of this project was to find out if it is possible to make a game for iPad where people would prefer to play that specific game with widgets compared to touch.

During one of the interaction tests we found a tendency for people to prefer lifting the widget, to activate the hover function (in the widget version), but did not prefer to lift their fingers from the screen to activate the same function (in the touch version). Because of this it was decided to look further into if there was any significant difference between what people prefer when they have to do the same action in a game either when holding a physical object or when using touch. Therefore we expanded the main test to also include a minor test which would make it possible to get some results on this.

A tutorial version of the game was made. The tutorial version was used to make the test participants familiar with the game before the actual test began. Besides that, we also used the tutorial version to test the preference in lifting fingers and lifting widgets against having a GUI button to activate the hover function. This means that each test participant tried four different tutorials of the game (Touch GUI, Touch Lifting, Widget GUI and Widget Lifting) and two different version of the main game (Touch interaction and Widget interaction).

Prior to playing the touch version of the game, the test participants played the tutorial versions of the same game with the same interaction method, meaning that they the two touch tutorials before playing the touch game whereafter they played the two widget tutorials before playing the widget game.

After the test participants had played the two tutorial version of the game for either touch or widget they were asked which of the two tutorials they preferred to play. This would indicate whether they preferred lifting or using a GUI button in order to activate the hover function.

The main test was conducted as a randomized complete block design, with a total of 32 test participants completing the test. During the entire test a video camera recorded the event. Each of the participants played both versions of the main game for about 4 minutes, depending on when the current play ended. They had unlimited time to play in the tutorials versions, but were told to stop whenever they felt comfortable with the interaction and understood the game. While the test participants played the main game, a facilitator held a semi-structured interview with them, to get some qualitative data about their preference.

The topics which were discussed during the semi-structured interview were:

- Where do you look on the screen while playing.
- What does the interaction with the game feel like.
- How good of an overview do you have of the screen while playing.

Besides the semi-structured interview, the game also logged some data about each of the test participants activity in the game for each of the game sessions. The data logged contained the following variables (with the following units):

- How long time did the game take (seconds)
- Amount of distance moved (in-game units)
- Amount of degrees rotated (degrees)
- How long time did the player hover (seconds)
- How many times did the player use the hover function (count)
- Amount of shots fired (count)
- Amount of shots that hit the opponent (percentage)
- Amount of damage dealt with projectiles (count)
- Amount of damage dealt with power-ups (count)

The setup

When the two test participants entered the room, they were asked to take a seat on two chairs located on either side of a table, which had an iPad located between the two participants (Figure 32). The facilitator took place in a chair at the far end of the table, where a computer was place on the table. This facilitator was also the one who held the semi-structured interview with the test participants during the test. On the opposite side of the facilitator a video camera was located, facing the table and focusing on the iPad. Behind one of the test participants, two additional facilitators were sitting with each of their computers, taking notes of what happened during the test and writing down the answers from the semi-structured interview.

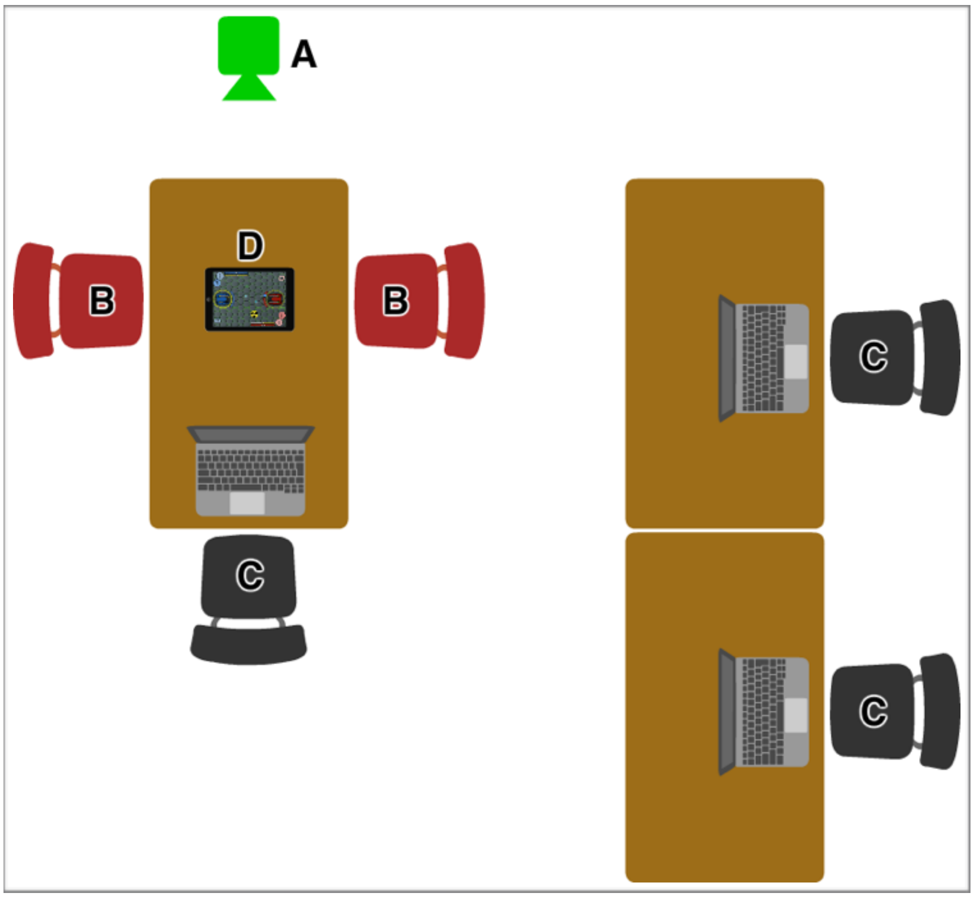


Figure 32: (A) The position and orientation of the video camera. (B) Two red chairs where the test participants were asked to take place after entering the room. (C) Three black chairs where the three facilitators were seated during the test. Each facilitator had a computer in front of them, used for taking notes etc. (D) An iPad located on a table between the two test participants.

The process

After the two test participants had taken place in their chairs, they were thanked for participating in our main test and were presented to who we were, in order to make them feel comfortable. After the presentation they were asked to sign a sheet allowing us to use the video footage and the data logged from the game as documentation for writing this report. When they had signed the sheet, they were asked about their age and what their main hand was.

The next step was to introduce them to the game they were about to play. A sheet with some instruction about the game was shown to them (see appendix A. Instruction Sheet to Hover Wars) and they were asked to read it all, and if they had any questions there were free to ask. When both test participants were done reading the instructions the first set of two tutorials were started. Before either of the tutorials they were introduced, orally by the facilitator, to how they could control the game with the given interaction method. After they had played both tutorials, they were asked which of the two tutorials they preferred to play. Finally they played the main game, with the same interaction method as the tutorials, for about 4 minutes. The test participants were told to “think out loud” while playing the main game. At the same time the facilitator asked some different questions to get a dialog going.

When they were done playing the main game with the first interaction method, they had to play the game with the second interaction method as well. The procedure for the second interaction method was the same as the first both in regards to playing the two tutorial version and the main game.

Finally, after having playing all tutorial version and both main games, the test participants were asked which of the two main games they preferred to play and elaborate on it. They were also asked if they had any other comments or thoughts about the test which they had just completed. When there were no more questions from neither the facilitator or the test participants, then the test participants were thanked once more for participating in the test and were told they could take a piece of fruit or some other snack as a thanks for helping us out.

7 Results & Discussion

This chapter will list the results of the experiment described in the previous chapter (6 Experiment). Each section will go through each part of the results (Lifting vs. GUI, Logged Data and Qualitative Data) and present the results. Then it will discuss what these results mean.

7.1 Lifting vs. GUI

During the test the participants were introduced to alternative methods of using the hover function in the game for both Widgets and Touch, namely lifting and GUI. They were then asked which of the hover methods they preferred. Their replies can be seen in Table 3.

Interaction Method	Prefers Lifting	Prefers GUI
Widget	27	5
Touch	17	14

Table 3: The number of participants preferring either using Lifting or GUI hover for the two interaction methods.

A Fisher's Exact test reveals that there is a significant difference between the two versions ($p < 0.05$). It is clear to see that when using widgets, players prefer the use of lifting the widget from the screen far more than using GUI. When using Touch, the preference is more evenly split. This suggests that the interaction of lifting from the screen has different appeal when using Widgets, than when using Touch. Therefore it would be interesting to look into what interactions are more acceptable when using widgets, than when using Touch.

7.2 Logged Data

In total, 652 data points were logged in the test, translating to 326 games played, each game being played by 2 players. Of those data points, 324 were from games played with Widgets, and 328 were from games played with touch interaction. As stated in the last chapter, each data point contains a number of different variables (see section 6.2 Preference Test). These variables represent the actions of a single user over the course of a game. Every game session had a different duration, so to make sure the variables are time-independent they must be divided by the total game time. If we for example take the distance moved and divide it by the game duration, the resulting variable can be described as "Distance Moved per Second". This is done for all variables, except the one called Accuracy, which is calculated by dividing the number of shots hit with the number of shots fired.

Table 4 shows the calculated mean values of these variables in the Widget version and Touch version respectively. It is also indicated which values had a significant difference. The difference in means between the Widget Version and the Touch Version are evaluated by a Two-sample Wilcoxon test.

Variable	Widget Version	Touch Version
Game Time (s)	**17.3659**	**16.2841**
Distance (units/s)	**4.3892**	**2.9700**
Rotation (degrees/s)	**46.7012**	**22.1108**
Hover Time (% of Game Time)	**0.1965**	**0.0525**
Hover Count (n/s)	**0.5673**	**0.1965**
Shots Fires (n/s)	3.9891	4.0309
Accuracy (%)	**0.4752**	**0.5483**
Damage Bullet (n/s)	**9.5531**	**11.4236**
Damage Powerup (n/s)	1.1401	1.4236
Damage Total (n/s)	**10.6932**	**12.5473**

Table 4: The calculated means of the data logged in the test. The values in parenthesis is the unit for the given value. Rows in ***bold*** *indicate statistically significant differences between mean values (p < 0.005)*

Table 4 shows that all variables, except for Shots Fired and Damage Powerup, shows a significant difference in mean value between the two versions of the game.

Discussion

Two of the variables could not be proven different between versions, and it is therefore assumed that the mode of interaction has no influence on these values. The explanation can be that the values are heavily dependent on the game design. In the case of Shots Fired, there is a set limit by the game on how often a shot can be fired. Assuming that the players fire whenever they can, this value will be maximized, leading to equal results. So even though these numbers cannot be used to prove a difference between the two versions, it might say something about the game design that people never stop shooting. The other value that has no difference is Damage Powerup. The number of damaging power-ups spawned during a game are too random for this value to make any sense, and the lack of significant difference in means show this.

From the data it can be seen that the means of Distance and Rotation are larger in the Widget version than in the Touch version. This can be seen as evidence that players are more active when using widgets, as they move more around the screen. The fact that Rotation is larger also suggests that rotating the object in this game is simpler when using widgets than when using touch.

The data shows that players utilized the hover function more often in the widget version than in the touch version. Both the Hover Time and the Hover Count means are greater in the widget version than in the touch version. This can be seen as evidence that using the hover function is easier in the widget version than in the touch version. As stated earlier, to hover in the widget version one has to lift the widget, and in the touch version the player must push a button to hover. Therefore, this data can be seen as evidence that the lifting interaction is easier than the button interaction. However, it is unknown how often the hovering was done intentionally by the player, or by accident.

The data also reveals small, but significant, differences in the mean of the variables Accuracy and Damage Bullet. Both values shows players being less effective with their in-game weapon, when using widgets. This could suggest that players find it easier to evade shots from the opposing player, which is further supported with Hover Time and Hover Count (players hover more often), and Distance and Rotation (Players move more around the screen). It could however also suggest that players are having a harder time aiming at the other player when using widgets.

7.3 Qualitative Data

After the test, 19 participants reported that they preferred using Widgets. 11 participants reported that they preferred using Touch. 2 participants were indifferent. These numbers reveals that the participants trended towards preferring widgets, but there were no significant difference between the two numbers.

During the tests, the players were encouraged to share their thoughts about the game, and about the different interaction method. Their comments were transcripted. In order to get an overview of what topics were mentioned most often, their comments were divided into categories. A selection of these categories can be seen in Table 5. Other categories were rejected for being infrequent (i.e. only mentioned by 3 people or fewer), or irrelevant to the project (i.e. comments not relating to the interaction method).

Comments about the Widget Version	Frequency		Comments about the Touch Version	Frequency
Focus on other players tank	20		Good overview of the screen	8
Easy rotation	12		Focus on other player's tank	8
Intuitive widgets	11		Focus on own tank	5
Occlusion	11		Difficult rotation	4
Intuitive lifting	10		Non-intuitive touch interaction	4
Interaction focused on one object	7			
Widget collisions is nice	6			
Widget too small/hard to hold	5			

Table 5: The frequency of different categories being mentioned during the test. Frequency is defined as the number of people mentioning something that relates to the given category. The names of the categories are descriptive of the kind of comment given. Players phrased their comments in different ways.

Discussion

The data show that many people expressed preference for widgets. 11 people stated that they found widgets to be intuitive, and easy to understand. Some of the reasons for this were that they liked the interaction being focused on one object, contrary to the touch version where the interaction is focused both on the tank, but also on the GUI buttons. This was stated by 7 people. Some people liked the fact that the widgets could collide with each other, leading to a different way of playing where pushing the other player was a part of the game.

On the other hand, some people found the size of the widgets to be a problem, namely that they were a little too small to hold properly. It is possible that this problem relates to the hand size of the given player, and whether or not it will be a problem for them. This suggests that it would be interesting to look further into the size of the widget.

All participants were asked to state where their focus and attention were during the gameplay. In the widget version, 20 people stated that they focused on the opposing player's tank. In comparison, only 8 people stated the same in the touch version. In contrast, several people mentioned that they looked at their own tank in the touch version, while nobody stated this in the widget version. This would suggest that when using touch people are more focused on their own tank in order to position it properly, while in the widget version the positioning comes more naturally, freeing up attention for watching the opponent instead.

A large group of participants stated that using widgets make it easier to rotate the in-game tank, than when using touch. This further supports the evidence from the logged data that players are more active in the widget version, since it is easier, and more intuitive, to move the widget around.

Finally, a group of participants mentioned the fact that the widget occludes a large part of the screen making it hard to see the objects on the screen. A solution to this problem could be to use smaller widgets that don't occlude as much of the screen, or using transparent material to make widgets. In any case, it is a problem that needs solving.

8 Conclusion

In this project, we presented a way to use tangible widgets for tablet games, and found data supporting the use of widgets as a viable interaction method for tablet games. Inspired by earlier research done, we designed physical widgets that would communicate several properties (i.e. position, orientation, identity and button push) to the touch device. By implementing a detection algorithm, it was possible to use widgets for designing games. In order to be able to test the use of widgets in a game-scenario, a game design was implemented that used the properties of the widget as integral parts of it's game mechanics. After having users play the game both with widgets, and with normal touch interactions, we asked them which interaction method they preferred, and recorded their answers. We also logged data from their playthroughs for analysis.

The data could identify several advantages of using widgets, most of them relating to the fact that movement on the screen was easier and more intuitive when using widgets, compared to touch. The data also suggested that players focused less on their own objects, and more on opposing objects when using the widgets. On the other hand the data also revealed drawbacks of using widgets. The fact that the widgets occluded a large part of the screen when being used was reportedly an issue for a number of participants. Also the size of the widget was a problem for some participants as they found holding on to the widget hard, and in some cases straining, due to its relatively small size compared to a hand.

Overall we conclude that widgets can be an interesting and useful means of interaction, but that there are issues with it's use. There are many roads to take this research from here. We could investigate different sizes of widgets, and the effect this will have on user experience. We could attempt to use different materials to make the widget, for instance transparent materials (to counter occlusion) or conductive material (instead of wrapping the widget in metallic tape). The widget used in this project used a makeshift button, though in the future it could be interesting to make a widget with an actual mechanical button. Finally, it could be interesting to look into different game designs, in order to fully verify the results found in this project, to see if the results can be replicated.

9 References

AppMATes. (2013). Retrieved May 17, 2014, from http://www.appmatestoys.com

Blagojevic, R., Chen, X., Tan, R., Sheehan, R., & Plimmer, B. (2012). Using Tangible Drawing Tools on a Capacitive Multi-touch Display. In *Proceedings of the 26th Annual BCS Interaction Specialist Group Conference on People and Computers* (pp. 315–320). Swinton, UK, UK: British Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=2377916.2377959

Bock, M., Fisker, M., Topp, K. F., & Kraus, M. (2014). Initial Exploration of the Use of Specific Tangible Widgets for Tablet Games. In L. M. Aiello & D. McFarland (Eds.), *Social Informatics* (pp. 183–190). Springer International Publishing. Retrieved from http://link.springer.com/chapter/10.1007/978-3-319-15168-7_23

Buxton, W. (1990). A three-state model of graphical input. In *Human-computer interaction-INTERACT* (Vol. 90, pp. 449–456). Citeseer. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.139.1700&rep=rep1&type=pdf

Fitzmaurice, G. W., Ishii, H., & Buxton, W. A. S. (1995). Bricks: Laying the Foundations for Graspable User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 442–449). New York, NY, USA: ACM Press/Addison-Wesley Publishing Co. http://doi.org/10.1145/223904.223964

Fullerton, T. (2014). Game Design Workshop: A Playcentric Approach to Creating Innovative Games, Third Edition. CRC Press.

Kratz, S., Westermann, T., Rohs, M., & Essl, G. (2011). CapWidgets: Tangile Widgets Versus Multi-touch Controls on Mobile Devices. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems* (pp. 1351–1356). New York, NY, USA: ACM. http://doi.org/10.1145/1979742.1979773

Rogers, S. (2010). Level Up!: The Guide to Great Video Game Design. John Wiley & Sons.

Schaper, H. (2013). Physical Widgets on Capacitive Touch Displays, Master's Thesis. In *RWTH Aachen University* (Vol. 1, pp. 1–118). Retrieved from http://hci.rwth-aachen.de/materials/publications/schaper2013a.pdf

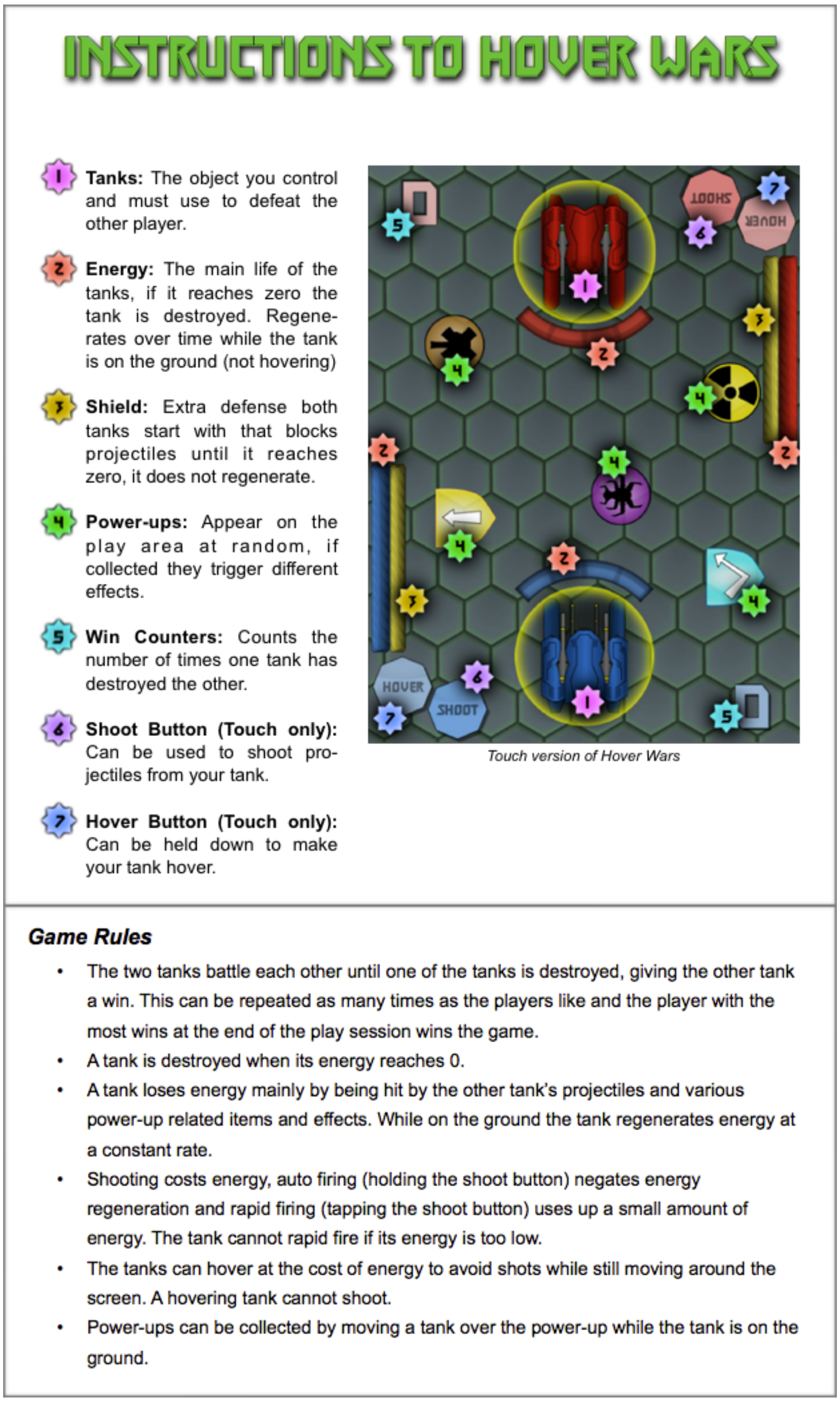
Schell, J. (2014). The Art of Game Design: A Book of Lenses, Second Edition. CRC Press.

Voelker, S., Nakajima, K., Thoresen, C., Itoh, Y., Øverg\aard, K. I., & Borchers, J. (2013). PUCs: Detecting Transparent, Passive Untouched Capacitive Widgets on Unmodified Multi-touch Displays. In *Proceedings of the 2013 ACM International Conference on Interactive Tabletops and Surfaces* (pp. 101–104). New York, NY, USA: ACM. http://doi.org/10.1145/2512349.2512791

Yu, N.-H., Chan, L.-W., Cheng, L.-P., Chen, M. Y., & Hung, Y.-P. (2010). Enabling Tangible Interaction on Capacitive Touch Panels. In *Adjunct Proceedings of the 23Nd Annual ACM Symposium on User Interface Software and Technology* (pp. 457–458). New York, NY, USA: ACM. http://doi.org/10.1145/1866218.1866269

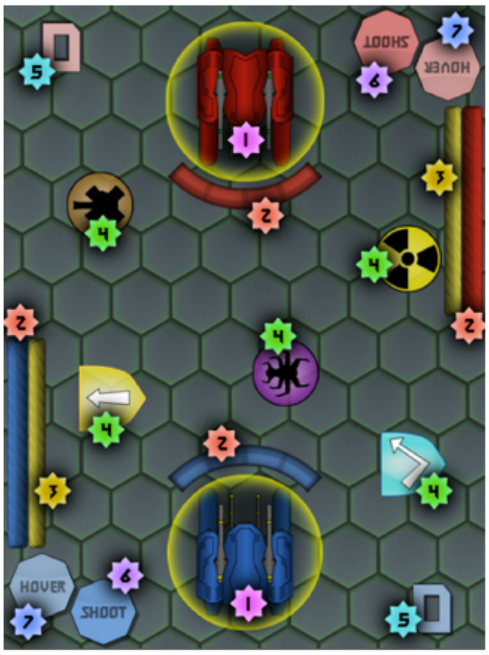
Appendix

A. Instruction Sheet to Hover Wars



INSTRUCTIONS TO HOVER WARS

1. **Tanks:** The object you control and must use to defeat the other player.
2. **Energy:** The main life of the tanks, if it reaches zero the tank is destroyed. Regenerates over time while the tank is on the ground (not hovering)
3. **Shield:** Extra defense both tanks start with that blocks projectiles until it reaches zero, it does not regenerate.
4. **Power-ups:** Appear on the play area at random, if collected they trigger different effects.
5. **Win Counters:** Counts the number of times one tank has destroyed the other.
6. **Shoot Button (Touch only):** Can be used to shoot projectiles from your tank.
7. **Hover Button (Touch only):** Can be held down to make your tank hover.



Touch version of Hover Wars

Game Rules

- The two tanks battle each other until one of the tanks is destroyed, giving the other tank a win. This can be repeated as many times as the players like and the player with the most wins at the end of the play session wins the game.
- A tank is destroyed when its energy reaches 0.
- A tank loses energy mainly by being hit by the other tank's projectiles and various power-up related items and effects. While on the ground the tank regenerates energy at a constant rate.
- Shooting costs energy, auto firing (holding the shoot button) negates energy regeneration and rapid firing (tapping the shoot button) uses up a small amount of energy. The tank cannot rapid fire if its energy is too low.
- The tanks can hover at the cost of energy to avoid shots while still moving around the screen. A hovering tank cannot shoot.
- Power-ups can be collected by moving a tank over the power-up while the tank is on the ground.