

CO3015 Computer Science Project

Dissertation

Title: Gaming with augmented human computer interface

Author: Anthony McQuilliam

Submission Date: May 2017

Department of informatics, University of Leicester

DECLARATION:

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s). Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s). I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

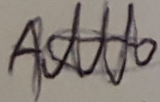
Name: ANTHONY MCQUILLIAM
Signed: 
Date: 1/05/2017

Table of contents:

1: Abstract	1
2: Introduction	1
3: Features	2
• 3.1: Tutorials and supplementary help	2
• 3.2: Game world	2
• 3.3: Enemies	2
• 3.4: Progression	3
• 3.5: Saving and loading	3
• 3.6: Game mechanics	3
• 3.7: Game visuals	4
• 3.8: Player features	4
• 3.9: Alternative input	4
• 3.10: Description of gameplay	5
4: Design, Implementation and Testing	10
• 4.1: Design	10
• 4.2: Video game implementation	11
◦ 4.2.1: Video game initial overview	11
◦ 4.2.2: View classes	11
◦ 4.2.3: Assets and world generation	11
◦ 4.2.4: A closer look at enemy AI	12
◦ 4.2.5: Other game details	14
• 4.3: Alternative input implementation	14
◦ 4.3.1: Overview	14
◦ 4.3.2: Camera and image processing	14
◦ 4.3.3: Image processing problems	17
◦ 4.3.4: Gesture recognition	18
• 4.4: Combining implementations	18
• 4.5: Automated testing	19
• 4.6: Human testing	19
• 4.7: Testing summary	20
5: Critical Summary	20
• 5.1: Social, commercial and economic impact	23
• 5.2: Personal development	23
6: Conclusion	23
7: Bibliography	24
8: Appendix	25

1: Abstract

The video game industry is an incredibly lucrative and constantly evolving market, seeking the next 'big thing', not just in terms of game ideas, but also how users interact with them. The purpose of this project is to create a video game that features a gesture recognition control system. Alternative inputs are not new to games, with systems such as the Microsoft Kinect being popular in the past. During this project I have assembled an image processing and gesture recognition software, that provides real time identification of gestures across a variety of user skin tones in order to provide an alternative input for a video game. In addition to this, I have created a video game that complements this input and by combining these two together the user can play the game purely through the use of gesture recognition. The final software system was tested through automated unit testing and through human driven testing, resulting in a rigorously tested product. Through the project several limiting factors were discovered and are discussed, detailing possible solutions and looking at future extensions that aim to solve these problems. Finally, this project looks at how a game using an alternative input system similar to this could be sold commercially, where it could compete with higher end products and where it could fall short.

2: Introduction

The global game industry brought in \$99.6 billion in revenue in 2016, with an expected growth to \$118.6 billion by 2019 [1]. Typical control inputs for these games are the hand held controllers, and for the PC, the keyboard and mouse. However, alternative inputs have grown in popularity in recent years, at the top end are technologies such as the HTC Vive's controllers [2], which allow users to interact with its virtual reality interface in a way that often mimics the user's own hands. Additionally, Microsoft's augmented reality Hololens [3] allows the user to interact with a projected image of the world. These devices have both changed the way users experience and play games. Certain genres of games tend to have more alternative inputs available to them, for instance, racing and driving simulation games often have inputs mimicking the control system of real cars, providing users with control over the steering wheel, pedals and gear sticks. Flight and space simulation games, tend to offer flight stick controllers, which attempt to simulate the flight stick of a real vehicle. In November 2006, the popular consumer electronics and video game company Nintendo released a console dedicated to an alternative form of input, the Nintendo Wii [4]. The Wii diverted from the use of a standard controller to a remote that, through an accelerometer and optical sensor, made use of motion sensing capabilities. While a large focus of the Wii was on sports and fitness games, it also brought games from genres that did not typically work well with alternative inputs, such as first person shooters and action and adventure games, providing the user with an alternate means of control. A particularly successful set of game franchises, were the Guitar Hero [5] and Rock Band [6] games, these music rhythm games allowed users to play along to popular songs and simulate performances. The user would accomplish this through either a standard controller, or more uniquely a controller specifically designed to resemble a musical instrument, for example a set of drums. Alternative inputs have additional uses to that of video games, modern smart phones for example, are capable of voice recognition which allows the user to control actions of the phone, such as searching the internet or organising their calendar purely through the use of their voice.

Due to the success of these previous consoles and games, alternative input schemes are always being explored. The software system I have designed looks at combining a standard video game with an alternative form of input, specifically, gesture recognition, in a way that is reliable and efficient. The aim was to create a system that could rival the use of the more traditional keyboard and mouse, in terms of ease of use, and effectiveness, while also creating a game that would not

necessarily be combined with an alternative input system, yet easily works with it. High end alternative input systems, such as the HTC Vive and Microsoft Hololens, can reach costs of several hundred, to several thousand pounds. So, in order to provide a realistic alternative to these high end products, the system makes use of a cheaper low end webcam, with the tested model costing just £15. It is through this low cost webcam that gestures are identified and used to control aspects of the game such as movement, aiming, attacking and triggering special abilities, as well as allowing the user to navigate through most menus.

3: Features

The following are the final features included in the system, divided into relevant groups. These are displayed with their name and a description of the feature. Later in the report, I will discuss how these final features relate to my initial requirements.

3.1: Tutorials and supplementary help

These features regard parts of the system that supply the player with knowledge on how to play the game and how more advanced features of the system work.

Tutorial screen: The user is able to access and navigate through a collection of tutorials which provide them with the knowledge of the basic features of the game, as well as explaining the more ambiguous aspects such as the traps and boosts.

Camera calibration guide: Supplied with the software is a camera calibration guide, which details which method of camera calibration is best for certain environments, how to proceed through each method of calibration and some basic trouble shooting tips on common problems that occur.

3.2: Game world

The game world provides the player and enemies a stage in which to interact, it contains several other objects that the player will come into contact with as detailed below.

Solid obstacles: Across the map are solid obstacles which the player is unable to pass straight through and must instead navigate around. These force players to think about future movement when engaging groups of enemies.

Traps: Traps spawn across the game map and upon collision with a player, causes a temporary negative status effect, such as slowed movement, weakened damage to enemies, a small amount of damage to occur to the player or instantaneous death. Forcing the player to be aware of their surroundings.

Boosts: Boosts spawn across the game map, they are the opposite to traps and upon collision with a player, cause a temporary positive status effect, either a small amount of health is restored or the player is granted a speed boost that will allow them to escape from enemies.

3.3: Enemies

These features concern enemies and enemy AI (Artificial Intelligence) which seeks to provide the user with a goal to the gameplay.

Enemies: The game features enemies that attempt to track, attack and kill the player's character. Each enemy has a distinct detection range at which they will begin the process of hunting down the player.

Varied enemies: The available enemies come in two types. The demon type, an enemy that attacks at close range, moves slowly and doesn't do much damage to the player but, has a large amount of health. The warlock type is a long ranged enemy with a small amount of health but, with faster movement and causes more damage.

Enemy path finding: All enemies make use of a path finding algorithm that allows them to navigate around solid obstacles, in order to track the player. This allows the player to use solid obstacles as cover from pursuing enemies.

3.4: Progression

In order to give the player a sense of achievement a set of features that facilitate a form of progression have been created.

Experience system: Players are able to earn spendable experience points from defeating enemies and surviving entire waves.

Upgrade system: Players are able to spend accumulated experience points to unlock and upgrade abilities, as well as upgrading their base stats: health, attack and speed.

Varying difficulties: Three levels of difficulty are supplied to the player, allowing them to start on the easy mode and as they upgrade their character, they are able to access harder difficulties. As the difficulty increases, enemies will gain health, damage and speed but, will also provide more experience points allowing the user to upgrade their character faster.

3.5: Saving and loading

Saving and loading allows the user to make progress, leave, and continue their progress at another time, it is a core set of features to most games.

Saving: The user is able to save their game, including their character's upgrade progress. This save file is stored locally and is saved in a way that does not allow the user to easily tamper with the file.

Loading: The user is able to load their saved games, allowing them to restore the progress they made to their character.

3.6: Game mechanics

The following are a set of rules and features that provides interaction with the state of the game.

Health: The player and enemies have health stats which once reduced to 0, cause entity death at which point it is removed from the game.

Win/Lose scenarios: If the player's health is reduced to 0, then their character is killed and they lose the game, sending them to a screen that allows them to quit or to restart. Should the player successfully defeat all enemies in all waves, they are taken to the victory screen and rewarded with

double experience points.

Pause: The user is able to pause the game at any time, allowing them to halt progress and continue when desired. The pause menu also provides a method of returning back to the game's menu without having to exit the game, although any experience gained in that game will be lost.

3.7: Game visuals

The game visuals concerns how the game looks, how the player's character, enemies, abilities etc. all appear to the user.

Visual representation: The game uses a 2D art style in order to visually represent what is happening.

3.8: Player features

To allow the user to interact with and actually play the game, the following have been included.

Movement: The player is able to navigate their character around the map in eight directions.

Aim: The user is able to aim the character, allowing them to choose the direction of their attack, when using the mouse they are able to achieve a full 360 aiming circle.

Attack: The user is able to use a basic attack, the purpose of which is to defeat enemies.

Abilities: The user has access to a range of abilities that can be unlocked and improved. The implemented abilities are: heal, an ability that heals the player; AOE (area of effect), an ability that causes an area of effect damage to enemies for a given time; slowdown, an ability that slows all the enemies movement for a given time; invisibility, an ability that renders enemies unable to detect the player for a given amount of time.

3.9: Alternate input

The following set of features all regard the alternative input system and what it provides to the software.

Image processing: Feed from a webcam is captured and processed into useful images that are then used to identify gestures.

Gesture recognition: User hand gestures are identified, in real time, and translated into a signal that is used to control aspects of the game.

Movement control: Users are able to control the movement of their character via hand gestures, allowing them to move in eight directions.

Aim control: Users are able to control the direction their character aims via hand gestures, allowing them to aim in eight directions.

Ability control: Users are able to trigger their various abilities via hand gestures.

Menu navigation: Users are able to navigate through certain menus via hand gestures.

3.10: Description of gameplay

This section gives an in depth look at how the game plays out, how the user may respond to various events and how the features all tie together.

Upon launching the game the users will find themselves taken to the main menu screen. Before starting the game they are given the chance to access the tutorial or load a previously saved game. From the main menu screen, if the user does not choose to load the game, the only other way to start the game is to click one of the “Start Game” buttons, one of these launches the game with the motion input and gesture recognition. The other does not, instead relying on the keyboard and mouse. Should the user choose to not use gesture recognition they are taken straight to the game menu screen. However, when choosing the gesture recognition option users are taken to the camera calibration screen, which allows them to configure their camera to their environment.



Main menu screen and camera calibration screen.

The user must now choose which method of calibration they wish to use, an explanation on how to use each method and which method might work best is provided to the user via the included camera calibration guide file. Once the camera has been set up the user must then perform the five finger gesture, this will take them to the game menu screen, and acts as a way to ensure that the user has correctly configured their camera.

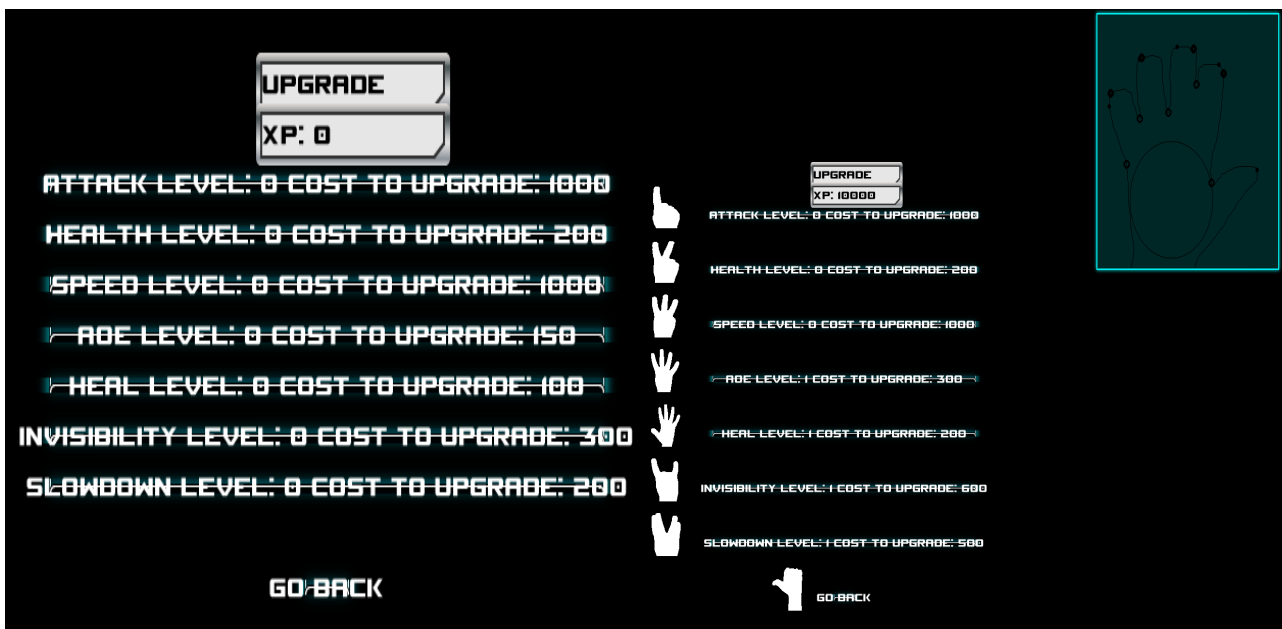
This now takes the user to the game menu screen, from here, all deeper views (if we consider the main menu screen to be the parent of the game menu screen, then the term 'deeper views' relates to all views that have the game menu screen as their parent) and their controllers will have at least two possible implementations, one for when the alternative input system is being used, and one for when it is not. The game menu screen allows users to access the game, via choosing one of the difficulty options, upgrade their character, save their game, return to the main menu or directly leave the game. When using the gesture recognition input, the user is able to navigate through various parts of the menus, as can be seen below, an image of the gesture is shown above or next to the button that it relates to. At this point for users with the alternative input turned on, a semi

transparent window is visible and shows the user their hand action, this allows the user to keep track of the gestures they are performing.



Images displaying the game menu screen for gesture recognition and keyboard/mouse control.

Games are saved by entering a save name, in the window to the right of the save game button, once this is done the user presses the save game button using the mouse, it will alert the user to whether their save has been successful. A save name containing a space will fail and the user will be alerted to this. The upgrade screen provides the platform for users to upgrade their characters, it displays their current accumulated experience points and shows them a list of upgrades that can be purchased. When an ability or base statistic, such as health, has been upgraded to the maximum the upgrade screen will reflect this.

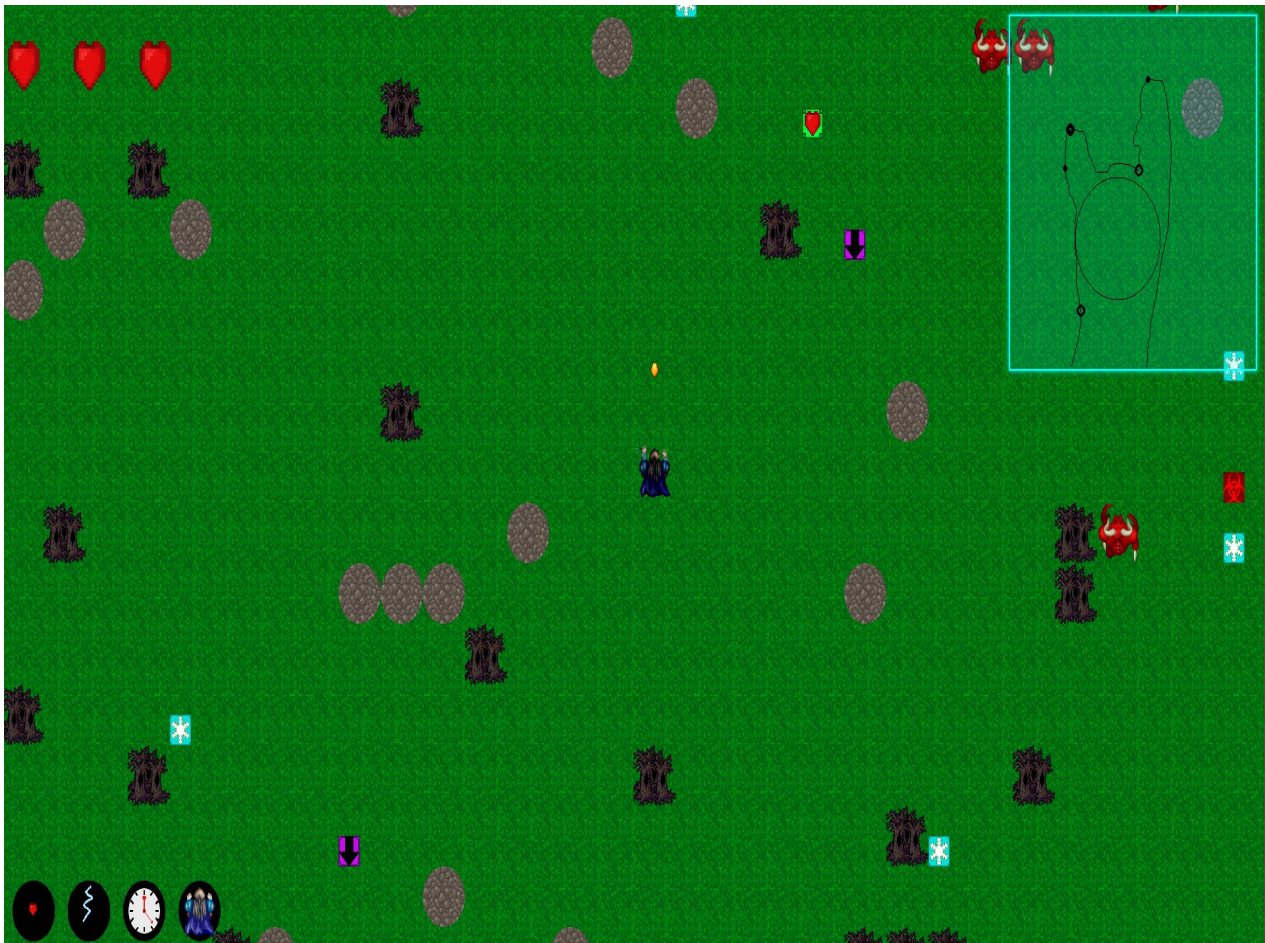


Images displaying the Upgrade screen in different states.

During the main game is where the majority of features occur. This screen works similar to the other screen classes, where required variables are initialised or created, before moving onto the repeated render loop, in which the render method is repeated each loop. It is in this render loop that the game scans for and acts upon the following: user input, enemy movement, gesture recognition processing and collision detection (for a more detailed run down of how this class works please see figure 1a in the appendix). The user will be placed in the middle of the map, surrounded by solid objects, traps, boosts and enemies. They will then have to navigate around the solid objects and traps whilst trying to defeat all enemies.

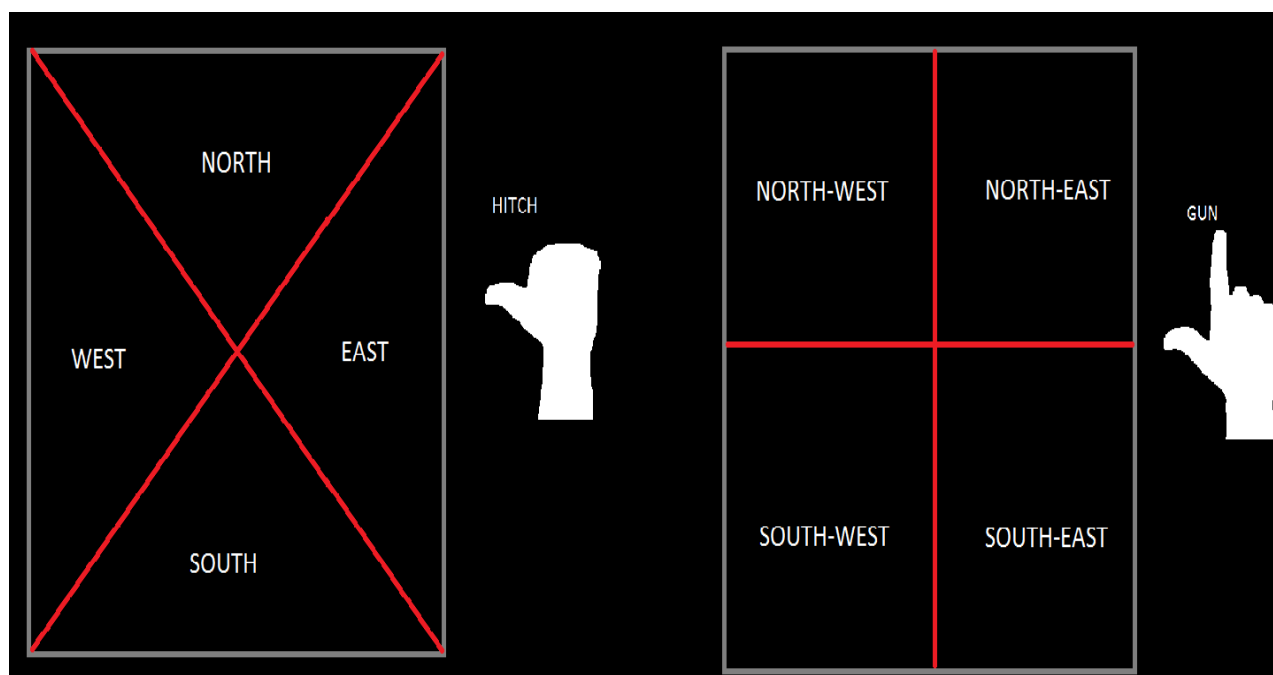
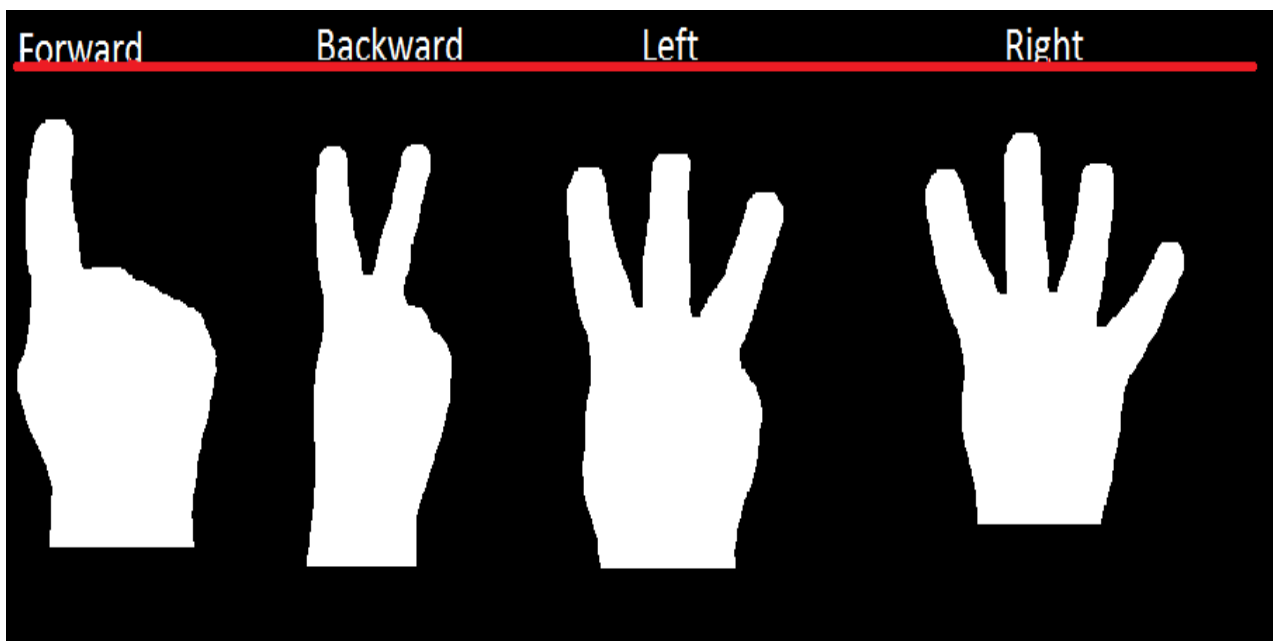


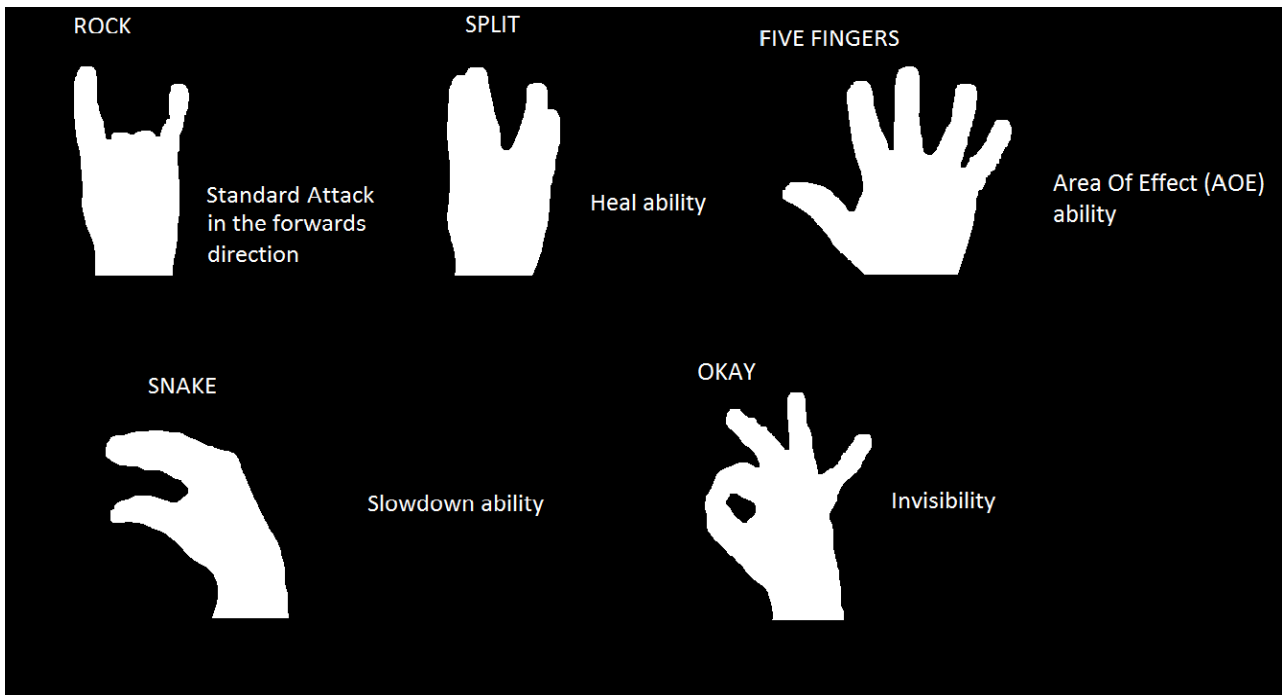
View in game: The trees and round rocks represent the solid objects, the player and the enemies cannot pass through these objects. The small coloured rectangles with symbols inside represent the traps or boosts. At the top left a visual representation of the user's health can be seen as hearts, this increases and decreases in real time as the user takes damage or heals. In the bottom left, the ability bar can be seen which displays a users abilities (if unlocked). When abilities are triggered, such as the AOE ability shown above, their icon in the ability bar becomes crossed through, this shows that the ability cannot be used again. Once the cool down timer for the ability has expired, the ability is 'recharged' and can be used again, at this point the ability's icon becomes the standard icon again.



View in game with gesture recognition: The game continues to show users real time feedback of their camera capture through a translucent window, this allows the user to continue to see the game world underneath and does not obscure the vision. The user can now control the game via gesture recognition, the tutorial on the main menu details how gestures are used. One, two, three and four fingers control player movement by moving forward, backwards, left and right (relative to the direction the player is facing).

The images below are taken from the tutorial screen and show the control scheme for gesture recognition.





Tutorial images, showing gesture control scheme.

Enemy AI has two stages, dormant and active. Each enemy has a integer value called 'detection range', this is the euclidean distance from the player, in pixels, to the enemy, once the player is within this range, the enemy is moved into an active state. The value of the detection range is dependant on the type of enemy and the difficulty being played on. While the player is outside of this range, the enemy remains still. Once in range the enemies are moved and rotated towards the player, until they are in their attack range (an integer value, determined by the type of enemy), at which point they will attempt to attack and cause damage to the player. This cycle of rotating/moving and attacking is repeated until either, the player or enemy has been removed from play, the player moves outside of the enemies detection range, or the player triggers their invisibility ability.

4: Design Implementation and Testing

The implementation of my software can be split into three subsections: one for the game, one for the alternative input and one for the combination of the two. Testing can be split into two subsections: automated unit testing and human testing. I will cover each section in detail.

4.1: Design

The design structure of the project, loosely follows the model view controller (MVC) approach, where models interact with a controller and the results are displayed to the user via views. However, in my implementation there are multiple controllers, at least one for each view, and then one to maintain and process instructions for core features of the game, such as ranged enemies, abilities and effects. The model components of the project have been split and organised into sub folders, where uncategorised models remain in the main model folder. There are sub folders for the Ability, Entity, Object, Projectile and Trap classes, each of these folders contains a parent class of the same name, and all other classes contained in the folders are extended from this class. This leaves me with a well organised and maintained structure. Outside of this MVC structure are five folders, Game, Flood fill, Motion, Path find and Test. The Test folder contains all code relevant to the

automated unit tests. The Motion folder contains classes that are required for the image processing and gesture recognition portion of the software. Flood fill and Path finding both contain their relevant algorithms, the use of which is detailed later in this report. Finally the game folder, holds the game class, this class is passed to all model, view and controller classes in the project, supplying the ability to draw to the screen. This gives me a total project structure that is organised and makes it easy to navigate to the correct class (see figure 2a in the appendix).

4.2: Video Game Implementation

I will first detail key aspects of the implementation of the video game.

4.2.1: Video game initial overview

The video game portion of my software has been created using LibGDX, an open-source, Java game development framework, which provides a powerful and unified API to be used [7]. The game's architecture is heavily based upon the MVC design pattern, where models contain the data and logic of the game, controllers process instructions to the models and change their state and views are the visual representations that are seen by the user. The code is split across two folders, Core and Desktop. Core contains all code for the project and is where all code written by myself is stored. Desktop is a launcher class (automatically created by LibGDX upon creation of a new project), that compiles the code in the Core folder along with various libraries to provide the final game. Given that there are several classes I have used that are provided by the LibGDX framework, to avoid confusion of what classes do and do not belong to myself, I will always clearly state that a class belongs to LibGDX when applicable.

4.2.2: View classes

All views in the game contain a reference to the games main class 'ArcherGame.java', this class is initialised every time upon starting up the program. The function of this class is to initialise two classes that will not be used at this point but will be used throughout the game by any classes that have a visual representation, the SpriteBatch and BitmapFont. The SpriteBatch is a class that allows the software to draw multiple sprites to the screen at once and the BitmapFont is the font that the game will use. Once this is done, the game passes itself and its initialised batch to the main menu screen, the first view the user will see, and will be seen every time the software is started. From here as the player interacts with the screens, the relevant controller will process the instructions. Each screen is attached to at least one controller which will extend the ClickListener class, provided by LibGDX, which will listen for click events that are called when the user presses the left mouse button anywhere on the screen, this mouse click will be translated into an input event that can be processed. The basic formula for each of these controllers is to get the actor (in this context an actor is an object of the 'Actor.java' class, provided by LibGDX and represents anything that can be interacted with by the user) that is attached to the click, test this against the list of buttons on the screen and perform the correct action.

4.2.3: Assets and world generation

The game features a large list of art assets which are loaded into the game through LibGDX's file loading class. Entities with animations, such as the player, the area of effect ability, the enemies, all use a sprite sheet, an image consisting of a smaller series of images called frames which represent an entities animation at a given point in time. In order to achieve this I have used code based on the provided LibGDX github animation guide [8], this code splits the sprite sheets into a two

dimensional array of images which is then converted into a one dimensional array where the frames are in the correct order for animation. This array is then used to create an instance of the LibGDX animation class. The animation is then called by a draw method each loop of the games render cycle, the draw method will check the entities state, such as whether a player is walking or attacking, and will play the correct animation associated with this state. When no action is detected, the entities current frame is set to the first frame of their walking animation, this gives a stationary appearance. For models that do not have animations such as the traps, boosts and the world map, a static image is loaded in as a LibGDX texture.

The game world, consisting of the player, the initially created enemies, the solid objects and the traps has to be generated. Generation of traps and enemies is very similar; while there are objects (traps or enemies) remaining to be generated, they first generate a random integer between 0 and 100, this is used to determine the type of object spawned, for instance whether a slow trap or a damage trap is spawned in. A random tile position is then generated, a check is made to whether this tile contains a solid object, if not then the newly spawned object is placed there, otherwise the tile position is recalculated.

Generating solid objects works slightly differently. Again a random integer between 0 and 100 is generated to decide the type of solid object to be spawned, and a random tile position is calculated, making sure that this position has not been used before and that it is not being placed on the player themselves. This is similar to the other two methods however, once all solid objects have been generated a breadth first search flood fill algorithm is applied to the game world, starting from the player's position. This algorithm checks that each tile that does not contain a solid object is reachable by the player, it does not allow for diagonal movement, stopping it from reaching tiles that would otherwise be inaccessible to the player (please see figure 3a in the appendix). Should this algorithm detect any unreachable tiles, it removes all solid objects and repeats the generation of solid objects from the beginning. Through the use of this algorithm, I can ensure that no boxes form causing the player or enemies to become stuck in unreachable areas.

4.2.4: A closer look at enemy AI

In order to move the enemies towards the player, the A* algorithm [9] was used. The premise behind this algorithm is to maintain two sets, called open and closed, where nodes to be considered are placed in the open set, and nodes that have already been considered and are not suitable are placed into the closed set. In order to achieve enemy path finding and movement, I have broken the overall algorithm down into two stages, the first where the path is generated using the A* algorithm, and then the second where the enemy is moved towards the player. To generate the path I initially proceed through the steps for A*, where I pass in an instance of the world class, the bounds of the world (its width and height), the tile size, and the start and end positions, where the start position is the enemies current position and the end position is the player's current position. The algorithm enters a while(true) loop, which is broken, with a break command, when the goal node has been placed into the path. First the cost (the distance from the start node to the current node plus the distance from the current node to the goal node) is calculated for each node in the open list, by summing together the distance from the start to the current and the distance from the current to the goal, the node with the smallest value is then selected. Next, the current node's top, bottom, left and right neighbours are checked for existence (an existing node is simply a rectangle within the map), we only check in four directions ignoring diagonal movement for the same reasons as the flood fill algorithm from section 5.13 (see figure 3a in the appendix). Each existing node is then checked under several conditions, if the node contains a solid object then it is immediately placed in the closed list, these tiles can never be considered as part of the path. If the neighbour node is contained

in the open list already, a check is made on its cost, if the cost of the neighbour node is higher than that of the current node then it is removed from the open list as the current path has a lower cost. Finally given that the neighbour node is not included in either open or closed lists, it is added into the open list to be considered on the next loop of the algorithm, additionally these neighbour nodes are then placed into a HashMap 'parents' that links a node with it's parent node (the node that comes before it in the path). This cycle repeats until the goal node has been found. From here the parents map is traversed backwards, starting from the goal node, retrieving the parent and placing them into a HashMap 'path'. This is the path that is used by the enemies for the second stage.

Step 1: Initialise required collections. Enter main loop.

Step 2: Look through the OPEN list of nodes, calculating each one's cost via $g(\text{node}) + h(\text{node})$, remove the node with the lowest cost from OPEN and add to CLOSED.

Step 3: Test if this lowest cost node is the goal node, if so break the loop and move to step 7.

Step 4: Test the nodes above, below and to the sides of the lowest cost node for existence.

Step 5: For each neighbour node that exists follow steps 5.1 through 5.4

Step 5.1: If the neighbour contains a solid object, add to the closed list and proceed to step 6.

Step 5.2: Else if the neighbour is contained in the open list, check if its movement cost is lower than the cost of current node, if not remove it from the open list. Proceed to step 6.

Step 5.3: Else if the neighbour is contained in the closed list, do nothing and proceed to step 6.

Step 5.4: Otherwise add the neighbour to the closed list and insert both it and the lowest cost node into the parents HashMap. Proceed to step 6.

Step 6: While loop repeats here, moving back to step 2.

Step 7: Starting from the goal node, traverse the parents HashMap backwards, generating a HashMap path to the start node.

Step 8: Return the path HashMap.

Description of A* algorithm that occurs in the Pathfind.java class.

In the second stage of enemy movement, the path has now been generated and the enemies can be moved towards the player by moving them so that their central point overlaps the central point of the next tile in the path, this continues until they are in range to attack the player. One of the issues encountered is that the player may continue to move once the path has been generated. To deal with this, every enemy has a marker of the node the player is currently on, after the path has been generated they also have a marker of the node the player was at when the path was generated, each time the method to move the enemy is called (each loop of the game screens render cycle), a check is made to whether the player is still on the same node, if not the path is recalculated. This results in a constantly updated system that will change direction to follow the player if required. In a system where the enemies would be tracking the player from much further distances, this method could be optimised to only updating the path when the player moves larger distances, however; even the highest distance the enemies track the player are close enough to warrant this design choice.

4.2.5: Other game details

A less technically impressive but important class is 'Configuration.java'. The purpose of this class is two fold, firstly it stores values for the player's stats and ability levels. When a game is loaded an instance of the configuration class is created and updated with the loaded variables, which creates a player that mimics the player from the loaded game. Secondly, this class stores variables that are used throughout the software, such as the cost to upgrade abilities and the stats of enemies on different difficulties. Although all values could have been stored locally in the classes that use them, the configuration class enables the ability to quickly modify critical game values, this was particularly useful during the testing stage of the project.

Several in game features cause a timed effect, the slowdown, AOE and invisible abilities all have time durations, additionally all abilities have a cool down period. 'Slow' and 'weaken' traps and the speed boost also have a timed effect. In order to maintain ability effects the 'AbilityController.java' class is used and the 'EffectController.java' class is used for maintaining timed effects caused by certain traps and boosts. These classes are also responsible for the instantaneous effects caused by abilities, traps and boosts as well. For the timed effect durations, the classes have methods which, when triggered, first check that this effect is not already active, if it is then the effect is ignored. An example of this would be if the player has collided with a slow trap, and is moving slowly, should they collide with another slow trap their character will not become any slower and the effect will wear off no later then it would have with the first one. This creates a game play mechanic of clearing as many of these effects as possible should the player come into contact with one. Should the method detect that the effect is not already active, it instantly applies the effect to the player, and then starts a thread for the effects duration, as these effects affect stats rather than graphics, manipulation can take place within the thread (no threads other than the main render loop may access graphic components due to the LibGDX framework, this is discussed later). Once the thread has reached the end of its execution the effect is removed from the player. For abilities, an additional thread is started monitoring the time before the ability can be used again, creating a game play mechanic that forces players to choose the best time to use their abilities.

4.3: Alternative Input Implementation

4.3.1: Overview

The alternative input has been created with the use of the OpenCV 3.10 library, an open source library that enables computer vision and machine learning [10]. And has been inspired by the work detailed in the journal by Yeo *et al.* [11], who provided a top level look at how a similar project worked, as well as some information that proved to be useful, such as recommended colour spaces. Specifically, the alternative input is a gesture recognition system, that not only detects which gestures are being used but, also, in some cases, where in the detection window they are being used. Feed from the webcam is processed which allows gestures to be recognised and translated into a String signal which is then used by parts of the game to enable gesture related control. Therefore, this alternative input can be broken down into two sections: the image processing, where frames captured by the user's webcam are turned into useful images capable of enabling detection of gestures, and the actual gesture recognition, where user's gestures are detected, recognised and converted into a signal.

4.3.2: Camera and image processing

The initial code for the camera setup has been very heavily based upon the code from the OpenCV

Java tutorial book [12], and its accompanying GitHub page [13], this is the section of the code that starts up the camera, grabs a frame from it, and allows it to be converted from OpenCV's Mat class into a image of .png format. Before the captured frame is converted from the Mat class into the image, it first must be processed, it is through a series of processing methods, that the frame is turned from a standard image into something that can be used by the detection system. This cycle of capturing and processing frames is repeated until the user exits the game or returns to the main menu. All of the following code for image processing takes place in the 'MotionInput.java' class. Once a frame has been captured, a method of thresholding is applied. There are three different methods that can be used and the decision of which to use is made by the user, they can base this choice upon the information they can find in the CameraCalibrationGuide.pdf file that is included with the software.

Method one works in an automatic fashion, and is based upon the idea of Simen Andresen [14]. Before any work takes place, the captured frame is converted from a BGR space (the default colour space) into a YCrCb colour space, this colour space was chosen due to its ability to distinguish between human skin colours [15]. Additionally, the separation of the luminance component (stored in channel Y) is useful in aiding with the issues of uneven illumination and bad lighting although, as I will detail later in section 4.2.3, these issues are still present in the software system. This colour conversion is done using OpenCV's built in colour conversion method (see 4a in the appendix). Once colour conversion has been completed, eight rectangles are placed on top of the camera input window (see 4b in the appendix). The image is then split into its three channels and then the intensity of each pixel in each channel inside the rectangles is calculated. This gives twenty-four average intensities per frame, eight per channel and each channel's averages are stored separately. The user then presses the T key to move the process on to the second stage. In the second stage the original frame is again split into its three channels. Here three lists are maintained, one for each channel. The program loops through the averages and uses the OpenCV built in thresholding method where each pixel in the image has an intensity value ranging from 0 to 255. All pixels below the supplied minimum threshold, which in this case is the average intensity, are converted into a black pixel. All those that are equal to and above the minimum threshold are converted to the supplied maximum value, which in this case is 255, resulting in a pure black and white binary image (see figure 4c in the appendix). By applying the thresholding method to each of the three channels using one of their respective averages, and storing these images in the correct lists, three lists remain, one for each channel, each containing eight images that have been processed with different values of minimum intensity (the average intensity calculated before). The final step in method one is to combine all eight images into three sets of one channel and then combine those three channels back into one. This results in a binary image where the user's hand and arm are all white pixels and background objects are black pixels. After this the contouring process can be applied.

Method two works in a semi automatic fashion, in which firstly the background is generated, a stage which requires the user to keep their hands out of the camera view. This generated background is constantly subtracted from the current frame, so as the background is generated this appears to the user as the image from the camera slowly fading to black. Once the camera feed is fully black, the user presses the T key, triggering the second stage of this method. For this stage, the generated background image is now constantly subtracted from all future frames, allowing the user to place their hand into the view of the camera and the hand will appear to be there with no background. This gives a system that only detects new objects entering the scene. Now the frame is converted into a greyscale image, meaning that it is now a single channel image, I could have converted to YCrCb like in the other methods but, only the Y channel would contain any useful information. The same thresholding method is applied to this image as in method one but, in this method, the user is

able to manually adjust the value of the threshold. Starting at one, they can increase and decrease the value until they get the image they require. However, unlike the similar stage in method three, this change in thresholding can occur at the same time as contour detection.

Method three is mostly manual, where the image is first converted into the YCrCb colour space, before it is split into its three channels. These channels are then processed by the thresholding method which is supplied with a changeable minimum value of one and a fixed maximum value of 255. The camera detection window now displays the first channel, from here the user navigates the channels using the left and right arrow keys and changes the minimum threshold value using the up and down arrow keys. They do this for all channels, ensuring that the hand is captured in white, and that there is no noise in the image. Once complete, the T key triggers the second stage where the values of the thresholding cannot be changed and where contouring is applied to the image.

All three methods of calibration use additional OpenCV methods to help with the processing. These additional methods are medianblur (blurring), erode and dilate. The first two are used to remove noise. The blurring method slides a kernel of a supplied size along the image, changing the colour of all pixels to the median value, which as it is a binary image must be either black or white. This results in stray white pixels being turned black and very little damage to the white hand pixels. Erosion works by sliding a kernel along the image, calculating a local minimum within the kernel and converting all pixels to this minimum, causing the black regions of the image to grow and the white to shrink. The result once more removes stray white pixels but, this method causes some damage to the white hand pixels (see figure 4d in the appendix). Dilation is the opposite to erosion and causes white regions of the image to grow and the black to shrink (see figure 4d and figure 4c in the appendix), it works in a similar fashion to erosion but calculates a local maximum instead.

At this stage, a single binary image now exists (see figure 4c in the appendix) and the contouring method is applied. The purpose of this method is to first detect the contour around the hand, this is achieved through the OpenCV function 'findContours' which uses the algorithm Suzuki85 [16], a modified version of the border following algorithm as described by Rosenfeld *et al.* [17]. Now the contour, which should appear as an outline of the hand, provided that the image has undergone the thresholding process correctly, should be visible. Now there may be several detected contours, caused by small amounts of noise or objects that temporarily enter the camera feed. In order to limit this to just the user's hand, the areas of all detected contours are calculated and a reference is made to the location of the contour with the largest area. Before moving on to calculating the convex hull of the contour, a few additional steps have to take place. The bounding rectangle around the contour has to be calculated, this will be used to ensure that only the user's hand is used at times when a large part of the arm may be present on screen (the user may be far away from the camera and may be performing a gesture at the top of the detection window). Furthermore, the largest inscribed circle has to be calculated, this is used in conjunction with the defect points to identify gestures, as well as checking whether or not a user's hand is present. To calculate the inscribed circle, we either check every fourth point inside the contour, or every fourth point inside of the top half of the contour (this is determined by the size of the bounding rectangle). Only the fourth point is checked in order to make this calculation significantly computationally cheaper. The distance between each fourth point and the closest part of the contour is then calculated, and the largest distance is recorded, as well as the central point. The distance and central point are then used to draw on the inscribed circle. In order to calculate the convex hull, the format of the data has to be converted from OpenCV's List<MatOfPoint> to OpenCV's List<MatOfInt>

Now the convex hull can be calculated, using the OpenCV method 'convexHull' which uses the Sklansky82 algorithm [18]. Sklansky82 is an amended version of the limited-applicability convex

hull finder algorithm [19], which had been limited to only working on polygons that contained no self intersections. Next the convex hull is processed to find the defect points, using the OpenCV method 'convexityDefects'. This method, when provided a contour and a convex hull, calculates and outputs a series of points to a `MatOfInt4` variable. A `MatOfInt4` is a type of variable that is provided by OpenCV and is similar to a list (in that it holds a list of variables) however, every four variables are linked together and every value stored in the `MatOfInt4` corresponds to one of four possible variables. If we call the `MatOfInt4` variable storage, then every value in `storage.get(j)` where $j \% 4 = 0$, is the start point, $j \% 4 = 1$, is the end point, $j \% 4 = 2$, the defect point and $j \% 4 = 3$ the depth (the distance of the defect point to the closest part of the convex hull). The `MatOfInt4` variable is then looped through and a check is now made to ensure that only defects with a significant value of depth are taken into account (those over 5000, a value which was determined by trial and error). There is now have a loop where each cycle of the loop deals with a different start, end and defect point and repeats for each detected set of points (see figure 4e in the appendix for a step breakdown of this cycle). This loop first calculates the angle of the defect, ensuring that only defects with an angle less than 90 degrees are used, as recommended by Yeo *et al.* [11]. A counter is used to count the number of defects and the (x,y) coordinates of start, end and defect points are stored, as well as the coordinates of the centre of the largest inscribed circle at that time. Checks are made on the positions of the points, such as whether they are to the left or right of the centre or above, below or inside of the inscribed circle. Also, the start and end point's position relative to the defect's position (above, below, left, right). Additionally, a check is made to see if the point is far away from the inscribed circle in terms of its x or y position. These values are the ones passed to the gesture recognition class to identify gestures, which returns a signal (see figure 4f in the appendix).

4.3.3: Image processing problems

Before going into detail on the gesture recognition class, I wish to identify some issues faced with the image processing. The first issue encountered was the issue of uneven illumination, this caused holes to form in the user's hands in the binary images, this was particularly noticeable when the hand was not held flat. Attempts to solve this problem were taken by moving the standard colour space into a YCrCb colour space thus, separating the illumination component. With this illumination component separated, I was able to use a higher value for the dilation method, causing the holes to become less obvious. Although, as can be seen when using method three's manual calibration, on the first channel there are several holes in the user's hands (see figure 4g in the appendix). Method two is the method of calibration least affected by the uneven illumination issue. In order to stop this from causing issues with the final image, the final image is also processed in the same way as the individual channels (blurred, eroded and dilated). The second issue that occurred happened when the lighting conditions were not favourable. This could be as a result one strong source of light coming from any direction or lack of light, and in this case all three methods are affected by this problem. However, in situations where this occurs, manual calibration is the best approach, as automatic relies on average intensity sampling which will significantly vary across the hand causing severe deviations in the detection. Unfortunately, although steps have been taken to try to limit the effect of bad lighting conditions, it ultimately could not be eliminated.

Another lighting related issue occurs when the lighting in the room suddenly changes, for example this could be due to increase in brightness outside or an internal light being switched on. When this occurs the user must recalibrate the camera each time, this can be problematic particularly in rooms heavily affected by sunlight, as this can lead to constant recalibration. Given more expensive hardware, it is possible that these issues could be completely eliminated, for instance, the Microsoft Kinect's camera provides depth information [20]. This depth information can be used for depth segmentation allowing for background removal and identification of objects that are only at a

certain depth and below. This depth segmentation process is a far more efficient system of camera calibration and the standard thresholding could then take place. Lighting still has some effect upon the Kinect however, it is significantly more resilient against small changes in light.

The system is built to be as resistant to these issues as possible, and allows the user to recalibrate the camera at any point, although a change of method would require them to return to the main menu of the game.

4.3.4: Gesture recognition

The gesture recognition class combines the information supplied to it (see figure 4f in the appendix) and uses this to identify gestures. The premise behind it is very simple, check the number of detected defects, perform a list of checks on possible gestures that can occur with this number of defects, return correct signal if one of these gestures is detected and return the 'noGesture' signal if all checks fail. In order to create a series of rules that would allow gesture recognition I captured several images of various user's performing these gestures while the system was displaying the defect, start and end points in real time (see figure 4h in the appendix). By analysing the similarities between these images I have been able to create a model of gesture recognition that works for user's with a range of hand sizes (see figure 4i in the appendix).

4.4: Combining Implementations

In order to merge the gesture recognition system with the video game two dedicated controllers were created, the 'MotionController.java' and the 'MotionMenuController.java' classes. The motion controller class manages gesture recognition used while playing the main game whereas, the motion menu controller manages gesture recognition used while navigating the menus. Both are similar in the way they work, the current screen performs a check each loop of its render cycle in its render method for the controller to detect and act upon gestures. For the motion menu controller, during this detect and act method, a check is made on all possible gestures (this is not the full list of gestures, but rather the ones that are used in the menus). If one of these gestures is detected a thread is started, lasting for one second. Should the detected gesture not be interrupted after the one second is complete the action associated to that gesture is performed (different for each screen). If however, another signal is detected during this one second counter, the thread is stopped and the action is not executed, instead either starting a thread for the newly detected gesture. Alternatively, in the case that a gesture is detected that is not related to the menu (including the noGesture signal), all running gestures are cancelled and no threads are called. The motion controller used whilst playing the game operates in a similar fashion although, when a gesture is detected no countdown thread is initiated before an action takes place but instead the action is instant. The code for the actions is essentially the same as the code from other controllers, with some minor differences being movement, in which the direction the player is aiming is first tested before moving in a direction that is relative to that aim direction (for example if the player is aiming East and uses the three finger gesture to move left, then they will move in a North direction).

An important part of combining these systems is allowing the user to see their gestures on the screen and they are able to use the camera detection window that pops up. However, this is solid and obstructs part of the user's view particularly when actually playing the game. To overcome this, a translucent window is placed in the top right corner (see figure 5a in the appendix), which copies the image from the detection window into the game. Conversion of these images on the fly takes up considerable resources, especially when the image is not binary. Thus, the user is encouraged to pause the game and use the camera detection window if they have to recalibrate, as the size of the

byte arrays of the images go from thousands and tens of thousands into the millions. Additionally, to save on computational power, rather than updating the image every cycle of the screen's render loop, a counter is maintained, this counter is incremented every cycle, and upon a certain value will then update the image and reset the counter.

4.5: Automated Testing

Automated testing was carried out using JUnit, Cucumber, and Gherkin scenarios, and is stored within the test folder of the Core folder of the project. In order to combine unit testing with the LibGDX framework first a headless client had to be setup [21]. This headless client makes use of the Mocking library Mockito, to fake graphics libraries and fake a client. As tests were being written it became clear that certain screen's constructors would have to be rewritten to allow me to pass into these mocked graphics components. These automated tests cover all of the features of the game, excluding any gesture recognition, and ensure that the algorithms such as the A* path finding and flood fill algorithms execute properly. Through the unit tests I was able to uncover and fix various issues, particularly with the path finding algorithm, in which the cost of movement was not being taken into account, interestingly the algorithm still appeared to work but not for the right reasons.

4.6: Human Testing

The human testing section was split into two phases, and used feedback from testers to influence certain decisions on balancing and to catch issues that were not being caught by the automated tests. The main advantage of this form of testing is that it allowed me to cover the gesture recognition input system as well as the mouse/keyboard input system. This meant that the system has been tested with a range of hand sizes and skin tones, to ensure that the software has maximum usability. In phase one testers were asked to play the game first with the gesture recognition, and then with the mouse and keyboard, while playing they were encouraged to say what they were thinking about the game, this information was recorded by me and the useful conversations were extracted and placed into the test form. After playing, the testers were asked to fill in the top half of the testing form. After this at the bottom of the form contained a section for an action plan, where I detailed any actions I was going to take regarding the feedback, and if no action was to take place (despite an issue being identified), then a reason why is given. Phase one was successful in both proving that the system was robust enough to be used with a variety of users and also in catching issues with certain gestures not being identified. It also highlighted that the most popular method of camera calibration was the automatic (method one) method, the third method was also used and interestingly, the second method was never used. Generally testers found that playing the game with the keyboard/mouse input was easier than the gesture recognition however, they found the gesture recognition more fun to use. A question of difficulty was raised, in particular, whether the difficulty of the game should be increased when using the keyboard/mouse input but, due to mixed results, this decision was put off until the end of phase two. A full summary of phase one was constructed (see figure 6b in the appendix).

Phase two, followed the same procedure as phase one, with a slightly modified form (see figure 6a in the appendix). Taking on feedback from the first phase, the form included the question of whether the difficulty should be adjusted for the different input schemes and whether there were any issues using particular gestures. The second phase of testing was unfortunately shorter than the first due to time constraints and difficulty in finding testers. However, the feedback was useful in determining that the difficulty of the game should not be adjusted and in identifying any more troublesome gestures, of which the three and four finger gestures were identified as being unreliable

(they would sometimes be detected but not consistently). Phase two did solidify the fact that most testers found the gesture recognition system to be more fun to use than the keyboard but, once again the second method of calibration was never used. It's worth noting that these tests occurred in the same place under similar lighting conditions and so I made the decision to keep the second method of calibration as I personally know there are some instances where that method works significantly better than the automatic system. A full summary of phase two was constructed (see figure 6c in the appendix).

4.7: Testing summary

Overall the testing has aided in ensuring that the finished software system is virtually bug free and that the features work as intended. Perhaps more importantly, that the game is interesting and fun to play. That the game mechanics work well with one another, the progression of the game's difficulties provides an interesting challenge and something for players to work towards. Of the two types of tests conducted, I personally value the human driven testing over the automated tests. This is due to feedback from people regarding various design choices, such as traps, boosts and experience earning. Most importantly, through human driven testing, I have found that the gesture recognition system is compatible with a variety of people.

5: Critical Appraisal

In order to evaluate my software, I will first look at what I originally set out to create, and what I have created. An extract taken from my project plan, "The overall aim of this project is to create a video game controller via the human body, motion tracking and image processing software, allowing for gesture recognition". Looking at this statement, on the surface, I can say that this is the final piece of software that I have created but, in order to supply a truly critical evaluation I will delve deeper into the components of the game; how they have been constructed, what could have been done better and what could be done next to evolve this project. To provide a brief description of my final software system the final video game is a top down two dimensional shooter, allowing the player to upgrade their character's abilities and base stats, such as health and speed through currency earned by playing the game and defeating enemies. It features varying levels of difficulties and consists of players attempting to defeat all of the enemies while navigating around obstacles and traps, using boosts and surviving waves of increasing difficulty in order to earn extra points. An alternative method of input is available in the form of a real time gesture recognition system that allows the user to navigate through the game's various menus and to control the player character's movement, aiming, special abilities and attacks.

By comparing the aims, objectives and requirements that I originally set out to accomplish, I can analyse the success of the project. Aspects of the game functionality were to include the ability to move the character in an eight directional system, as well as aiming in any direction. Both control schemes achieve the ability to move the character in eight directions, albeit in a slightly different manner from one another. However looking at aiming, mouse control gives the user a very fine control over the character's aim, allowing them to aim in any degree in a circle, whereas the gesture recognition is locked to aiming in one of the eight main directions (north, east, south, west and the four diagonals). The initial plan for aiming using gesture recognition was to allow the user to perform a gesture (thumb side gesture). When this gesture was detected the central point of largest inscribed circle would be taken and by calculating the angle between this central point and the central point of the camera detection screen, it would allow the user to aim the character just like the mouse. However, in practice this system was incredibly unstable and unusable for the following reasons; the inscribed circle often bounces around the hand and rarely remains still, additionally

some angles were unachievable as the user's hand would be required to be outside of the detection window. Unfortunately this resulted in having to reduce the functionality of aiming with gestures in order to give the system accuracy. The next attempt of the system was to use one gesture to allow aiming in all eight directions however, this faced a similar problem where the character would bounce between directions. The final implementation sees the user using two gestures for aiming, one for the standard north, east, south and west aiming, and another gesture to aim in the diagonal directions. While this gives the system a very good degree of accuracy it does cause a significant change in how the game plays when using both systems and unfortunately creates a system in which the keyboard and mouse are superior to use.

The next features to look at are the attacking, ability, enemies, health, visual representation, saving, loading, varying enemies, currency, upgrading and increasing difficulty. All of these features have been implemented to a good standard and work the way they were intended to on both control systems. The enemies have been implemented particularly well, from their first incarnation of simply constantly moving towards the player's position, ignoring solid objects, to a system that incorporates detection distance and an advanced path finding algorithm, allowing enemies to track the player whilst navigating obstacles. One criticism over the enemies could be that they do not interact with the traps and boosts spread across the map and they do not make any conscious effort to avoid them, whereas the player must avoid the traps otherwise they face a penalty. The design rationale behind this was that these traps would have been laid out by these enemies and so would know how to move without disturbing them. While in the context of this game, this is a common game play mechanic featured in several well known games however, it is worth noting that due to the well formed structure of the path finding class, any future developer would have no trouble in adding the list of traps (a list that is maintained by the world class) to be avoided. Additionally, on the topic of varying enemies, due to the easily extensible nature of my 'Entity.java' class, additional enemies are easy to add, future developers would only be required to supply the art and link values from the configuration class to this new enemy class, the entity class will handle drawing and maintaining the sprites.

Two requirements that were initially listed did not make it into the final game. The first of these is the concept of 'boss' enemies, these enemies were intended to be very hard versions of enemies that would appear as the final enemy of the game, defeating them would result in a victory and would reward the player with lots of points to use to upgrade their character. However, due to time constraints these enemies were never featured, it would be very easy for any future developer to add these enemies, the only real work required would be in changing how the enemies would spawn in this final boss wave. The second requirement that did not make it into the final game is the concept of scoring. Originally the scoring system was intended to give the player some feedback on how successful they were at the game however, this system was removed purposely when the concept of rewarding the player extra points upon victory was introduced, the scoring system seemed redundant and provided little measure of success and so was ultimately removed.

The next set of requirements concern the gesture recognition system, and they were menu navigation, ability control, movement control, gesture recognition and motion tracking. All together the system is well functioning and relatively robust, and has been able to achieve most of the requirements originally expected. As previously detailed (with character aiming) the motion tracking was not as successful as I would have liked, and has been limited to provide a system with accuracy. Looking at the gesture recognition there are some critical evaluations to be drawn. The system only works with the right hand, now while the act of forming a gesture is far less intricate than say writing, user's would still prefer to use their dominant hand when controlling a game character. If there was any future development, allowing users to choose which hand they wanted to

use, would be one of the key features I would want to add, then the detection system would use the right checks to determine the gestures as it does now. Another implementation of this would be to allow both hands to be considered at once, this would allow significantly more gestures and more control over the character. Another criticism of the gesture recognition is that the user must take care to keep their hand away from their body, as the body, head, face, arm, everything is taken into consideration, which ties into one of my earlier objectives to use a Haar cascade to remove the face. However, this would not have aided with the rest of the body, and so has resulted in the user having to keep their hand far away from their body. A final note on the gesture recognition, the user must replicate the gestures exactly as they were recorded, there is very little leniency towards sloppy gestures, even something relatively innocent such as the user tilting their hand can result in an inability to correctly identify gestures.

Continuing the analysis of my software, I will look at the overall design of the product. I am very happy with my design approach of creating easily extendable parent classes for: traps, abilities, entities, objects and projectiles. This means that if future development was to occur, that it would be simple to add significantly more content to the game, a large list of varying enemies and scenery could be added, with minimal tweaking and work. The configuration class was another good design choice, allowing developers to quickly change variables that are used throughout the game, some applying to enemies, some to the player and some to the game itself. This means that balancing can happen quickly and effectively, condensing a potentially long process into a shorter one. Looking at the negatives of the design, the largest critical point I could draw is how much of the code occurs during a screen's rendering loop, this is incredibly computationally inefficient, due to being locked into using only one thread. Unfortunately, this design choice is forced due to the use of the LibGDX framework, which does not allow any external threads (threads outside of the render loop) to access graphics and audio libraries. Because of this limitation, I have had to use a system of scanning within the render loop, such as scanning for collisions, scanning all enemies and checking if they are dead, etc... While I have had some success in using additional threads, such as monitoring ability durations (e.g. how long the player remains invisible), this has had to be done using a system of boolean guards that are scanned for and toggled by the render method, these extra threads must also not have any effect on the graphics or audio components.

The human driven testing, I believe, was carried out very well. However, I would criticise the number of phases and the number of testers per phase. If there were any future development plans, I would suggest a much longer, more intense testing period, in which significantly more feedback is collected and used to influence design choices. A good example of how the low numbers caused an issue, a question of whether the difficulty should be changed depending on which control scheme was used was proposed by one of the testers, early on in the first phase. Future testers were asked their opinion on this and although overall the feedback was to leave the difficulty as it was, this should not be thought of as a well informed decision considering the small sample size. In a more commercial environment, where given more time and resources, this number would be expected to be significantly higher, and therefore any decision made from that information would be more justified.

This leads me on to a interesting evaluation of what I would do differently in hindsight. Concerning the use of the LibGDX framework, although it provided a very useful and large API, with many useful classes, I feel that the limitations that it imposes through its lock of the graphics library in particular, perhaps outweighs the usefulness. I feel that now I have the understanding of the complex OpenCV library, I would have had the time to write my own game engine and given the rather simplistic nature of the game I do not believe this would have been too challenging and would have given me full control over how the resources of the program were managed.

5.1: Social, Commercial and Economic Impact

Due to this very low price of hardware required for this software to fully function, I believe that this project, and projects similar to it could easily be converted and extended into a commercial product. It could tap into a market of people who wish to experience an alternative input system, but who cannot afford the higher end products. The gesture recognition system used in this project could also be applied to other systems, not only games. Household appliances such as televisions could also be controlled via gesture recognition, interactive displays and menu navigation are all possible applications. Socially, alternative input systems may allow people to communicate easier. A form of communication, texting, using a mobile device relies on the user typing out messages to send to another user however, most current smart phones allow the user to now construct these messages via voice recognition. Economically, other than the obvious gaming industries addition to the economy, this project has very little effect outside of a commercial purpose.

5.2: Personal Development

Throughout the development of this project, I have gained valuable knowledge into the design process of creating a video game, as well as learning about the different alternative input systems that are currently being used and that could be used. From a technical stand point, I have learned a large array of features and algorithms from the OpenCV library, as well as the A* path finding algorithm enabling AI paths. Although unsure on my own future career, these technical skills are particularly useful in the field of robotics, machine learning and optimal path finding in systems, where constantly changing environments is an area of particular interest. Other skills that I have been able to develop is the ability of breaking down a seemingly daunting task into smaller manageable tasks that together formed this project.

6: Conclusion

For this project, I have developed a video game, with a variety of interesting features and game mechanics, that can be controlled via a relatively robust gesture recognition software. The game features a well made implementation with lots of easily extendable classes, allowing for further development if required and a easily configurable path finding system, based upon a solid, well known and well used algorithm. The gesture recognition is built upon a trusted and well tested library aimed at computer vision, using algorithms that have been optimised for the best possible performance. Although, I have had to compromise on some the design features in order to provide a more stable system. Issues with the system have been identified and unfortunately although attempts to reduce their effect on the system has taken place, they still exist. Future work conducted on this project, or projects of a similar nature should aim to tackle these illumination related issues, as well as expanding the gesture recognition system into using more than just one of the user's hands.

My system provides a video game with a host of extendable classes, allowing for rapid addition of content, as well as a well structured design, and very clear commented code, making any future work very easy to perform. The gesture recognition system is able to identify user gestures in real time and with a small amount of work, allows future developers to add more gestures in a system that is strict enough to distinguish between different gestures but is flexible enough to be used with varying hand sizes and skin tones. I believe that I have achieved my original aims and objectives and have created the piece of software that I set out to create.

7: Bibliography

- [1] <https://ukie.org.uk/research>
- [2] HTC, <https://www.vive.com/uk/product/>
- [3] Microsoft, <https://www.microsoft.com/en-gb/hololens>
- [4] Nintendo, <https://www.nintendo.co.uk/Wii/Wii-94559.html>
- [5] Activision, <https://www.guitarhero.com/uk/en/game>
- [6] Harmonix, <http://www.rockband4.com/>
- [7]: Mario Zechner, 2013, <https://libgdx.badlogicgames.com/features.html>, viewed 20 April 2017.
- [8] LibGDX github team, Github Repository, <https://github.com/libgdx/libgdx/wiki/2D-Animation>
- [9]: P. E. Hart, N. J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, 1968, IEEE transactions of systems science and cybernetics, vol. Ssc-4, no. 2
- [10] <http://opencv.org/>
- [11] H. S. Yeo, B. G. Lee, and H. Lim, "Hand tracking and gesture recognition system for human-computer interaction using low-cost hardware", 2015 Multimedia Tools and Applications, 2015 74: 2687. DOI:10.1007/s11042-013-1501-1
- [12] Luigi De Russis, Alberto Sacco, OpenCV Java Tutorials Documentation, Release 1.0, December 28, 2016, <https://media.readthedocs.org/pdf/opencv-java-tutorials/latest/opencv-java-tutorials.pdf>, Chapter 3.8, pages 20 – 22.
- [13] Luigi De Russis, Github repository, <https://github.com/opencv-java/video-basics/blob/master/src/it/polito/teaching/cv/VideoController.java>
- [14] Simen Andresen, August 12th 2013, <http://simena86.github.io/blog/2013/08/12/hand-tracking-and-recognition-with-opencv/>
- [15] P. Sebastian, Y. V. Voon and R. Comley, "The effect of colour space on tracking robustness," 2008 3rd IEEE Conference on Industrial Electronics and Applications, Singapore, 2008, pp. 2512-2516. DOI: 10.1109/ICIEA.2008.4582971. Print ISBN: 978-1-4244-1717-9 CD-ROM ISBN: 978-1-4244-1718-6 Print on Demand(PoD) ISBN: 978-1-4244-1717-9
- [16] Suzuki, S. and Abe, K., *Topological Structural Analysis of Digitized Binary Images by Border Following*. CVGIP 30 1, pp 32-46 (1985)
- [17] A. Rosenfeld, A. C. Kak, Digital Picture Processing, 2nd ed., Vol. 2, Academic Press, New York, 1982
- [18] J. Sklansky, Finding the convex hull of a simple polygon, Pattern recognition letters, 1982, Vol.

1(2), pp. 79-83, ISSN: 0167-8655. DOI: 10.1016/0167-8655(82)90016-2

[19] J. Sklansky, Measuring concavity on a rectangular mosaic. 1972, IEEE Trans. on Computer C-21, 1355-1364.

[20] Microsoft, <https://msdn.microsoft.com/en-gb/library/hh438998.aspx>, revised 2017

[21] H. Pellikka, <http://manabreak.eu/java/2016/10/21/unittesting-libgdx.html>, october 21st 2016

8: Appendix

Figure 1a:

A sequence of steps showing the process of how the InGameScreen.java class works.

Step 1: Initialise instances of the core classes, 'World.java' and 'Player.java'. Setup the camera, the heads up display. Additionally the controllers are setup, one for the game (to handle user input), one for the camera (to handle camera scrolling as the user moves around, allowing the user's character to move around the screen without going out of view), one to monitor collisions between various actors in the game, one to handle any alternative input. Finally one for each type of enemy, ranged and melee.

Note: Step 2 and beyond are repeated each time a call to the render method is made.

Step 2: All generated actors are drawn to the screen.

Step 3: The camera controller monitors for user input.

Step 4: Additional Stages are drawn in, such as the heads up display and if motion is enabled, the detection window.

Step 5: If the motion is enabled, then the program checks a counter, if this counter has reached a certain value it will update the detection windows image, otherwise it increments the counter. This is done to boost performance of the software.

Step 6: A check is made to whether the state of the game is running or paused, if the game is paused then the pause window is drawn on and the class returns to Step 2.

Step 7: Given that the game is running, checks are made on the enemies, to update the number of enemies on screen if required.

Step 8: Actors with a limited range, (Actors that belong to the 'Projectile.java' class), are moved along and a check is made to whether they have exceeded their range. If so they are removed.

Step 9: The motion controller checks whether it can detect any gestures, if so it acts upon them.

Step 10: Enemies are now tested to see if they should be moving towards the player, and checked to ensure that they have at least one unit of health, otherwise they are removed.

Step 11: User input and active collisions are scanned for.

Step 12: A check is made to see whether the player still has over one unit of health.

Figure 2a:

Image displaying project structure.

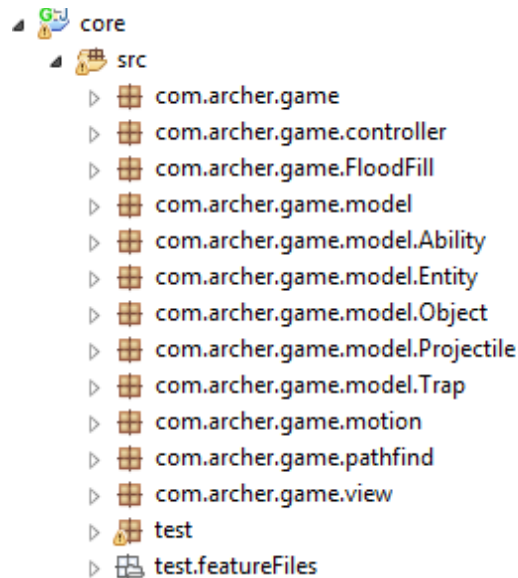
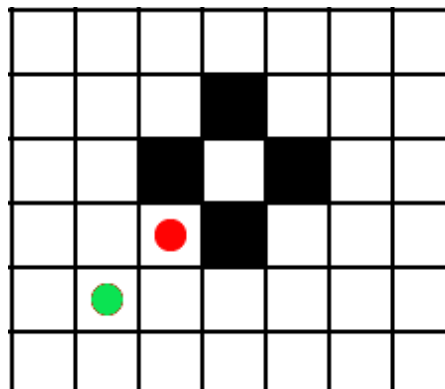


Figure 3a:

Although the player is capable of diagonal movement in the game, the below image displays why the flood fill algorithm does not consider diagonal movement.



Given that the player is the red circle, the player can freely move to the green circle south-west of it. However it cannot navigate into the tile north-east due to placement of the (black tiled) solid objects. Thus explaining why the flood fill algorithm cannot consider diagonal movement.

Figure 4a:

image from http://docs.opencv.org/3.1.0/de/d25/imgproc_color_conversions.html, displaying how colour is converted from RGB to Y Cr Cb

$$\begin{aligned}
Y &\leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \\
Cr &\leftarrow (R - Y) \cdot 0.713 + \text{delta} \\
Cb &\leftarrow (B - Y) \cdot 0.564 + \text{delta} \\
R &\leftarrow Y + 1.403 \cdot (Cr - \text{delta}) \\
G &\leftarrow Y - 0.714 \cdot (Cr - \text{delta}) - 0.344 \cdot (Cb - \text{delta}) \\
B &\leftarrow Y + 1.773 \cdot (Cb - \text{delta})
\end{aligned}$$

$$\text{delta} = \begin{cases} 128 & \text{for 8-bit images} \\ 32768 & \text{for 16-bit images} \\ 0.5 & \text{for floating-point images} \end{cases}$$

Figure 4b:

Method one of the camera calibration in action, after the background has been removed but before thresholding has taken place.

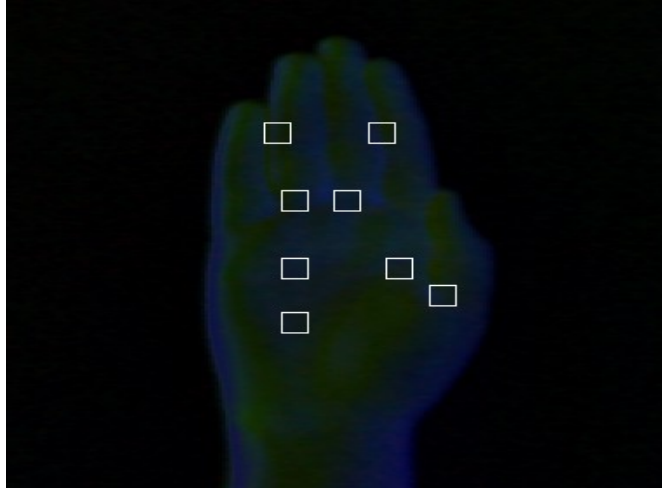


Figure 4c:

Frame after being converted to a binary image, blurring, erosion and dilation has taken place at this point.



Figure 4d:

Frame after being converted to a binary image, blurring and erosion has taken place at this point. Notice that before dilation the fingers appear visibly thinner and wrist has been reduced.



Figure 4e:

A sequence of steps showing how the values used in gesture detection are determined. This occurs each time a new frame is processed.

Note: To avoid confusion with steps 6 and 7, please note that each start, end and defect point has

similarly named variables which are incremented for the same reasons. Such as defectsToTheRight, startToTheRight and endToTheRight, are all incremented when the point's x position is found to be greater than the centre's x position.

Step 1: All variables are initialised to 0.

Step 2: A check is made on the depth value of each defect, only allowing defects with a certain depth to be considered.

Step 3: The angle between the start and end point for each defect is then calculated, only defects with angles below 90 are to be considered.

Step 4: The defect, start and end points are all drawn onto the frame.

Step 5: The x, y positions for the defect, start and end points are all stored.

Step 6: For each start, end and defect the following checks are made:

Step 6.1: If the point's x position is greater than the centre's x point. Then the points to the right variable is incremented by one (such as defectsToTheRight++)

Step 6.2: If the point's y position is less than the centre's y point – the centres radius, then the points above palm variable is incremented by one.

Step 6.3: If the point's y position is greater than the centre's y point + the centres radius, then the points below palm variable is incremented by one.

Step 6.4: If the point's x position is greater than the centre's x point + it's diameter or less than the centre's x point – it's diameter, then the points far X variable is incremented.

Step 6.5: If the point's y position is greater than the centre's y point + its diameter or less than the centre's y point – its diameter, then the points far Y variable is incremented.

Step 7: For each start and end point the following checks are made:

Step 7.1: If the point's y position is less than its defect's y position then the points above defect variable is incremented. Otherwise the points below defect variable is incremented.

Step 7.2: If the point's x position is less than the defect's x position then the points left defect variable is incremented

Step 8: These variables are passed into the gesture recognition class.

Step 9: The signal is retrieved from the gesture recognition class.

Figure 4f:

Code extract of passing variables to gesture recognition.

```
gestureRecognition.inputNumberOfDefects (numberOfDefects) ;
```

```
gestureRecognition.inputStartVariables (startAbovePalm, startBelowPalm,  
startToTheLeft, startToTheRight, startFarX, startFarY, startAboveDefect,  
startLeftDefect, startBelowDefect) ;
```



```
gestureRecognition.inputEndVariables(endAbovePalm, endBelowPalm, endToTheLeft,
endToTheRight, endFarX, endFarY, endAboveDefect, endLeftDefect, endBelowDefect);

gestureRecognition.inputDefectVariables(defectsAbovePalm, defectsBelowPalm,
defectsToTheLeft, defectsToTheRight, defectFarX, defectFarY);

gestureRecognition.inputDoubles(defectAngle, totalAngle, largestDistance);

signal = gestureRecognition.getGesture();
```

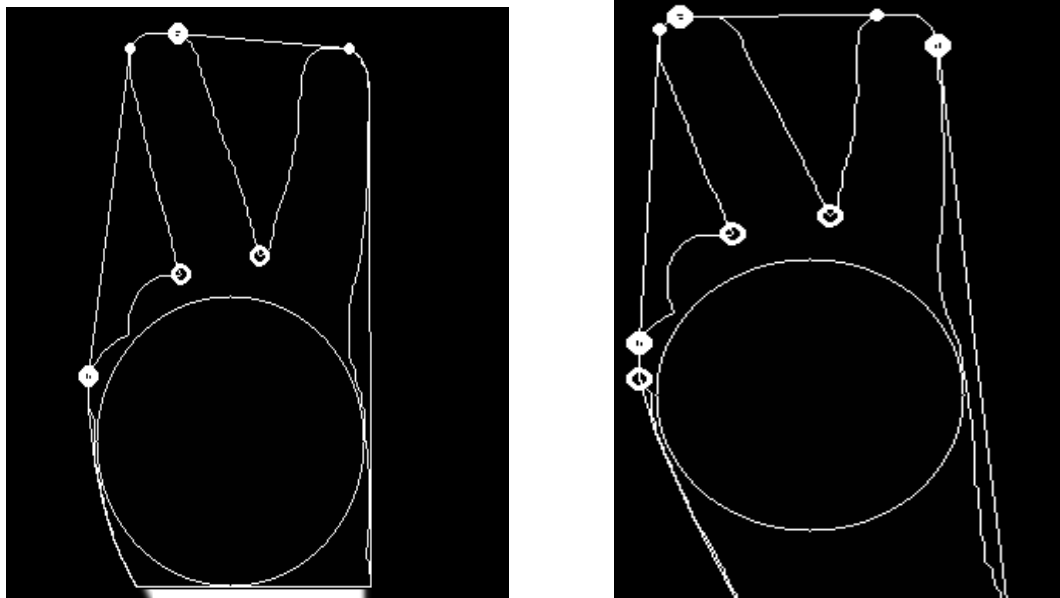
Figure 4g:

Holes in the hand caused by uneven illumination. From channel one or the Y channel.



Figure 4h:

Several images captured when trying to get a pattern to identify the two finger 'peace' gesture.



The above two images are the results of capturing the gesture as it was being performed. During

actual game play, these two variations of the same gesture may occur and so the model for gesture recognition has to take this into account.

Figure 4i:

Code extract from GestureDetection.java, showing how the three fingers gesture is identified when the number of detected gestures is three.

```
// test number of defects, gesture can occur at 3, 4 or 5 defects
if (numberOfDefects == 3) {
    int check = 0;

    // all points should lay above the palm, its possible that one
    // defect may dip
    // inside the palm
    if (defectsAbovePalm >= 2 && startAbovePalm == 3 && endAbovePalm == 3) {
        check++;
    }

    // start and end points should all be above the defects
    if (startAboveDefect == 3 && endAboveDefect == 3) {
        check++;
    }

    // there should be at least one of each point each side
    if (defectsToTheLeft >= 1 && defectsToTheRight >= 1) {
        check++;
    }

    if (startToTheLeft >= 1 && startToTheRight >= 1) {
        check++;
    }

    if (endToTheLeft >= 1 && endToTheRight >= 1) {
        check++;
    }

    // test check
    if (check == 5) {
        return true;
    } else {
        return false;
    }
}
```

Figure 5a:

A snap shot of the in game window detection screen, notice that the background is clearly visible behind the window, but that the user's hand is also clearly noticeable.

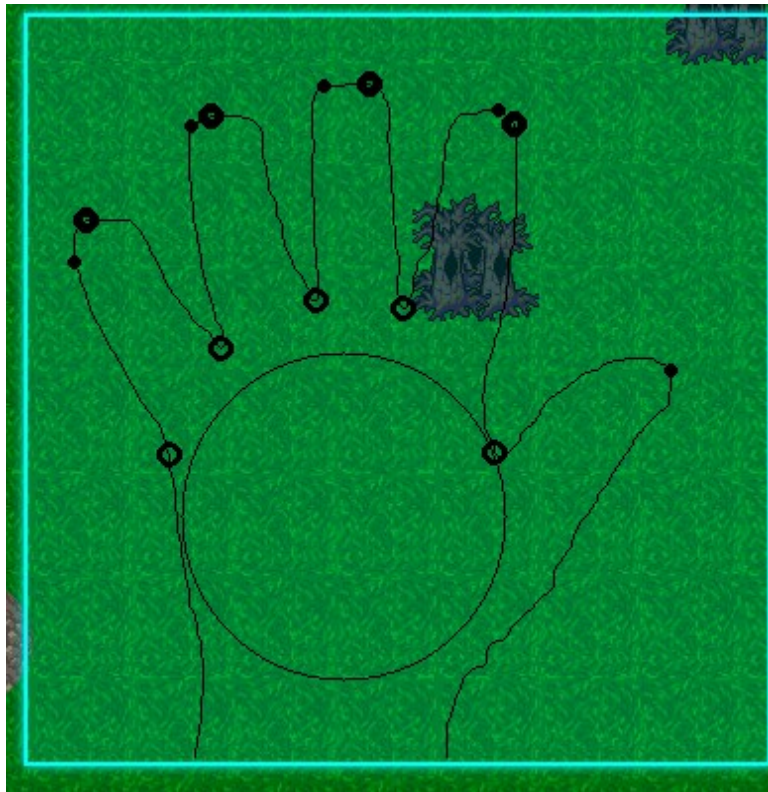


Figure 6a:

Test form for the first phase of testing

Testing Form

Brief explanation of what was tested:

Testers have been asked to carry out several tasks in the game using both motion and keyboard/mouse control. While playing the game using the motion input, testers will comment on the ease of use of the calibration system, any comments on the gesture recognition system as they navigate through the menus and play the game, they will also comment on how they feel the game works with the gesture recognition. While playing the game using the keyboard/mouse controls they will comment on how they feel the game works as a whole. All comments made by testers during testing will be recorded by me for use later on. Once complete testers will be asked to fill in the top half of the form, which has a list of questions they can answer. The bottom half of the form will be filled in by me using the recorded comments and will contain any actions I intend to take based on the feedback.

Part 1a: Questions (For Testers)

Which method of calibration was used:

Did you find the calibration system easy to use:

Were you able to navigate through the menus:

Were you able to move the character using gestures:

Were you able to aim the character using gestures:

Were you able to trigger all abilities using gestures:

Were you able to attack enemies using gestures:

Which control system did you find easier:

Which control system did you prefer:

Part 1b: Additional comments (For Testers)

Please write down any additional comments you have:

Part 2a: Recorded feedback

Part 2b: Action plan

Testing form for the second phase of testing.

Testing Form

Brief explanation of what was tested:

Testers have been asked to carry out several tasks in the game using both motion and keyboard/mouse control. While playing the game using the motion input, testers will comment on the ease of use of the calibration system, any comments on the gesture recognition system as they navigate through the menus and play the game, they will also comment on how they feel the game works with the gesture recognition. While playing the game using the keyboard/mouse controls they will comment on how they feel the game works as a whole. All comments made by testers during testing will be recorded by me for use later on. Once complete testers will be asked to fill in the top half of the form, which has a list of questions they can answer. The bottom half of the form will be filled in by me using the recorded comments and will contain any actions I intend to take based on the feedback.

Part 1a: Questions (For Testers)

Which method of calibration was used:

Did you find the calibration system easy to use:

Were you able to navigate through the menus:

Were you able to move the character using gestures:

Were you able to aim the character using gestures:

Were you able to trigger all abilities using gestures:

Were you able to attack enemies using gestures:

Did you have any trouble with any of the gestures, if so please state which:

Which control system did you find easier:

Which control system did you prefer:

Do you feel that the difficulty of the game should be adjusted when using the keyboard and mouse:

Part 1b: Additional comments (For Testers)

Please write down any additional comments you have:

Part 2a: Recorded feedback

Part 2b: Action plan

Figure 6b:

My recorded summary of the first phase of testing

Testing Phase One Summary

The first phase of testing has been completed and has been relatively successful with few issues encountered. As predicted there were some issues with the camera calibration system, particularly in environments with unfavourable lighting conditions.

The following gestures caused some testers problems: Thumb Side, Three Fingers, Four Fingers, Snake and Okay. For Thumb Side, Three Fingers and Four fingers, the detection was unreliable, sometimes being picked up and sometimes not. For Snake and Okay the tester could not get them to work at all.

There was a discussion about the difference in difficulty between the motion and keyboard control systems. Naturally keyboard and mouse gives the user a finer control system to work with. Some testers felt that the difficulty of the game should be increased when playing with the mouse and keyboard, however the idea was split down the middle with half the testers agreeing.

The general consensus is that while the game is easier to play using the keyboard and mouse, the motion input control system is more fun to use. The most common method of calibration was method one, with one instance of method three being used. In all cases method two was either not required or not suitable for use.

Action Plan:

I will investigate problems with the identified gestures. I believe that a combination of the user not being strict with their finger positions and the detection system being too strict with how gestures are recognised are the cause of problems. This is backed up by the fact that no two testers had troubles with the same gestures. The question of changing the difficulty for the keyboard and mouse controls is a harder choice to make, due to the finer control system of the keyboard and mouse it makes sense, however the feedback was mixed. I have decided to do nothing with the difficulty for now and will ask the next phase of testers their opinion on the difficulty.

Figure 5c:

My recorded summary of the second phase of testing

Testing Phase Two Summary

The second phase of testing has now been completed, and has been very successful

with only some minor issues being identified. The issue of difficulty was addressed directly in the questions, again leading to a mixed result. Additionally this phase of testing had one less tester than phase one due to time constraints

The following gestures were identified as causing some issues: Three fingers, Four fingers. In all cases where there were issues these gestures were unreliable in their identification, sometimes being recognised and sometimes not.

The question of difficulty was built into the questions part of the test form this time, however, the majority of the testers were in favour of not changing the difficulty, with one tester commenting that perhaps the alternative input game should be made easier and the keyboard controlled game should be left as it is.

The general consensus is again that the mouse/keyboard control scheme is easier to use, but that the alternative input is more fun to use. The methods of calibration were split half and half between method one and method three, with method two not being used at all.

Action plan:

I will investigate issues with the identified gestures, although I believe it to be a problem of bad camera calibration causing weak hand identification leading to poor gesture recognition. Given that more people have been against increasing the difficulty of the game for the mouse/keyboard control scheme I will leave the difficulty as it is.