De Montfort University

Annihilation Intelligence (A.I.)

A 3D 1st Person Horror Mobile Phone Game

Final Report

By Simon Chiu

Computer Games Programming

P14133969

Abstract

Annihilation Intelligence is a 3D 1st Person Horror mobile phone game, developed in Android Studio for the Android OS (operating system). This has been developed as a foray into creating and developing an entire game, as the Computer Games Programming course does not venture far from development, and focuses on the coding and programming of a game rather than the game as a whole.

Annihilation Intelligence was developed using Android Studio, utilising j-PCT-ae for 3D, coded in the Java programming language, as well as XML for the activity layouts. 3D objects were created in Autodesk Maya 2016, along with simple textures, and music and sound effects were created in FL Studio, Bfxr, or recorded.

Table of Contents

[Introduction 4](#_Toc477465835)

[Background and justification 4](#_Toc477465836)

[Method 5](#_Toc477465837)

[Activities 6](#_Toc477465838)

[Activity Life-cycle 6](#_Toc477465839)

[Loading/ Saving Data 7](#_Toc477465840)

[Flow of data through Activities 7](#_Toc477465841)

[Intents 7](#_Toc477465842)

[Singleton Classes 8](#_Toc477465843)

[Why data flow is important 8](#_Toc477465844)

[Write up away from Computer 9](#_Toc477465845)

[Separating the Game Activity and Game Surface View 9](#_Toc477465846)

[j-PCT World axis 9](#_Toc477465847)

[Designing in Maya and loading the scene 9](#_Toc477465848)

[Joysticks 10](#_Toc477465849)

[Touch-sensing 10](#_Toc477465850)

[Buttons 11](#_Toc477465851)

[Text 11](#_Toc477465852)

[Separate levels (combine with scenes?) 11](#_Toc477465853)

[Collisions 12](#_Toc477465854)

[Keeping UI relative to screen 12](#_Toc477465855)

[Inventory and using items 12](#_Toc477465856)

[Text Buffer 13](#_Toc477465857)

[Changing floors 13](#_Toc477465858)

[Asset Management 13](#_Toc477465859)

[Memory Management (IMAGES) 14](#_Toc477465860)

[Number pad 14](#_Toc477465861)

[Instruction Activity 14](#_Toc477465862)

[Puzzles 16](#_Toc477465863)

[Conclusion 17](#_Toc477465864)

# Introduction

## Background and justification

There are a few reasons for the choice of project. Firstly it was surrounding a system of great interest: Mobile phone games. In today’s society, mobile phones are, and continue to be, a large part of everyday life. It has long evolved from a simple communication device, to a utility for work, a platform for playing games, and even for scheduling a person’s day to day life. Besides being a large part of life, the mobile phone game market has also been constantly expanding, as it is a gaming platform where the console is easily accessible to people (as most people own a mobile phone) and the games are generally quick to play. People can play simple games to pass the time while travelling, or while waiting.

This has been an area of games creation which is extremely interesting. Under the Computer Games Programming course, students develop some experience with mobile games, namely from the IMAT 2608 Mobile Games module, where students developed simple 2D mobile phone games in Android Studio. However, this was limited to 2D games, and this project presents a chance to create a 3D mobile phone game, and explore difficulties which may arise from 3D mobile phone game development. With the increase in the power of mobile phones, 3D games have become increasingly popular for mobile phones, with large video games companies who originally make games for home-console systems also joining to mobile phone development.

Another reason for this project is the general interest in games creation, notably independent game development (Indie games). Indie games have become more popular over the years, which did not rely on large graphics or visuals, but instead focused more on story or game play to entice players. There are many cases of successful indie games, for example the *Five Nights as Freddy’s* series by Scott Cawthorn, which has been discussed in the previous literature review. But another example is *Stardew Valley* by Eric Barone. Developed over four years, all parts of the game, from programming and design, to art and composition, were all done by Eric. This is an inspiring story for indie games developers, showing Eric’s passion for this game, and his hard work spent on the game.

As well as the creation of a game, the genre, Horror, was also a point of interest. In recent years, Horror games have been a popular choice for indie developers, as using jump scares or the use of atmosphere in the game was a simple, yet effective method of engaging the player. During the literature review, the term *Horror Vacui* was explored, and the fear of something which might exist, yet we cannot necessarily see it. This was very interesting, and the idea of trying to create a Horror game while avoiding the cliché uses of jump scares or sudden movements was intriguing.

# Summary of Project

## Project Plan

The interim deadline showed promise in the use of j-PCT-ae for 3d development, as well as showed the detection of touch-screen inputs to control the character. For the final version, a project structure had to be formed, similarly to how games were structured in the IMAT 2608 Mobile Games module from second year. This separates the project into Activities, Classes and Views. Activities contain the different ‘screens’ of the game, such as the start-up screen, the options screen, etc. Classes contain the java classes we use to define objects, for example the player, certain objects in the world, etc. Lastly the Views are how the game is displayed. The game will use a surface view to render and draw objects.

### Repository/ Source Control

Before continuing from the interim deadline, an online repository was set up to hold the project, allowing better management of the project, as well as recording changes to the project, and time stamps of when the project was updated. Appendix 1 shows the list of commits to GitHub, an online repository service to upload projects in progress, as well as allow easy access for other people to a project. Each of these commits represent a change to the project, such as new activities and classes, fixes to bugs and errors, as well as changes to assets, documentation, or even testing in a new branch.

### Activities

The currently planned activities are:

* Splash Activity – This will be the starting screen, which is used to load certain parts of the game, or prepare files for the game
* Menu Activity – This is the main menu the player will see, and will allow access to other activities
* Game Activity – The game itself, this will then have a surface view to render and play the game in
* Options Activity – This will have the editable options for the player, such as changing the volume of music or sound effects, or changes to controls of the game

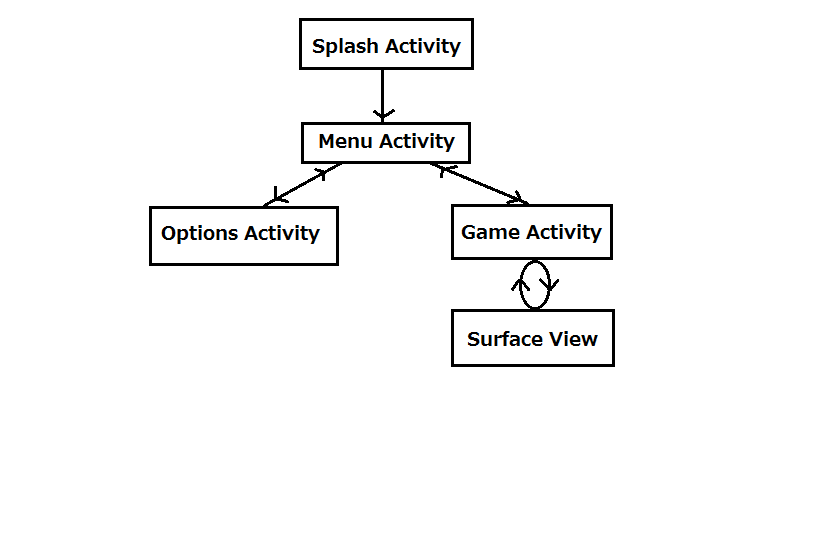


Fig.1 – Planned activity flow. The Surface View runs alongside the Game Activity

#### Activity Life-cycle

Starting up the game enters the Splash Activity. This splash screen allows us to start loading parts of the game, hiding this from the user with a loading screen. From here we enter the Menu Activity. This is the main hub from which we can access other activities. The Splash Activity will not be accessible anymore. Trying to return to the previous activity (the Splash Activity) from the Menu Activity will instead close the game, showing a toast message to confirm if the user wants to quit the game.

We can branch to either the Game Activity or the Options Activity. In the Options Activity, the user is able to set certain settings, such as the volume of the music and sound effects, as well as an option to reset these to default settings. Unless the user confirms these settings, leaving this activity and returning to the Menu Activity will return the settings to the last used setting. The Game Activity is linked to a Surface View, which is how the 3D objects are rendered to the screen, as well as updated over time. The interaction between the Game Activity and Surface View is important, for example when the player touches the screen, the Game Activity takes this input, then passes it to the Surface View to process, and when the Surface View requires access to the Game Activity, such as playing a sound effect, it can run a function to access the activity. When the game ends, the Surface View will return this to the Game Activity, and it will return to the Menu Activity.

#### Flow of data through Activities

Certain data will be reused in different activities, such as the volume of the music, which must be consistent throughout the activities, and must use the volume set in the Options Activity. When moving from one activity to another, we can set certain data to also be passed, by changing the Intent, which is essentially the new activity. By doing this we can also recover data from a child activity back to its parent.

For example, the options data will be loaded from the Splash Activity, and passed to the Menu Activity. When accessing the Options Activity, the data is again passed, and changing any of the options and leaving the Options Activity will return the new data to the Menu Activity, which will update its local version of the data, as well as the text file where it was originally loaded from.

Java does not allow the use of Pointers in ways that C++ does. We cannot create a pointer for an integer for example, and pass the pointer through the activities. Because of this, each activity has a local version of any passed data, and starting or ending an activity requires setting the current activity with any new data. There are pointers in Java to an extent, but this is more for accessing the data in an Activity Layout.

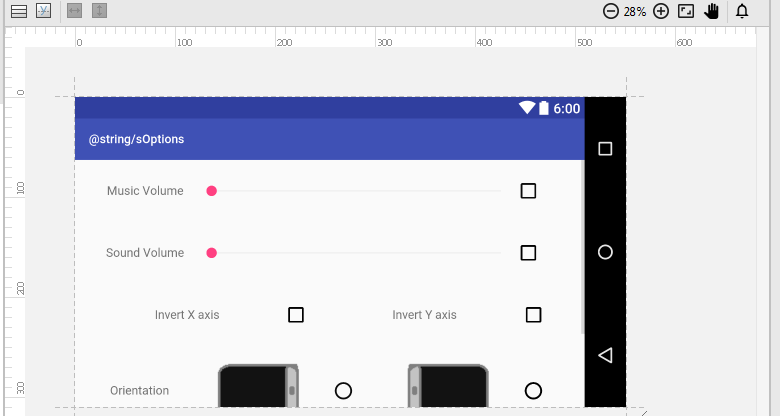


Fig. 2 – The Activity Layout for the Options Activity

In Fig.2 we can see the activity layout for the Options Activity. The various check boxes, radio boxes and seek bars can be accessed with Java pointers, searching for the ID of a widget. Using this we can access and change these widgets from the Activity code. Widgets such as buttons are given an onClick variable, which becomes a function in the Activity code.

#### Why data flow is important

The setup of data to be used throughout the app is extremely important. We save loading times by not loading the same data over again in different parts of the app, but instead by loading all the data in the Splash Activity, acting as a loading screen to then allow the rest of the app to run more smoothly. Players do not want to be waste time reloading data, especially on a device with limited computational power. Similar to early console games, efficiency is key to developing a fast-running game.

#### Loading/ Saving Data

Certain data, such as the volume of music or the orientation of the screen should be saved, so if a user has changed the settings, they can keep those settings when they run the app again. To do this, certain data is saved to a text file, and this can be loaded from and saved to. In the Splash Activity, this data is loaded by reading the text file and parsing each line in the text file. If the text file or directory does not exist (such as when running the app for the first time, or if the data is somehow deleted) a default text file will be created with the default settings.

Similarly when saving data which has changed from the options, data is saved as individual strings, with backup data if it cannot be written properly to the text file. Having default data to fall back on is imperative to avoid errors, or potentially crashes.

## Development

### Activities

As a foundation to build upon, the activities were created first, and the flow through the activities was developed. Although each activity was an empty activity, they were the shell for the entire game, meaning additions to each activity was easy to add, as opposed to trying to rearrange activities should a change occur, which happened with the introduction of the Instructions Activity, which will be explained later on.

### Singleton Classes

As well as the activities as a foundation of the project, the flow of data between these activities had to be set up. The use of intents was explained in the Project Plan; however, this is limited to certain data types, such as integers or strings. Problems arose with the need to pass music and sound, which is loaded from the Splash Activity, to the rest of the project. To fix this problem, a singleton class was created: a class where only one instance of it exists. This means any data set to this instance can be accessed by other activities and each activity will not have a local version, but instead share a global version of the class. The Media class was created, which is first used to load music and sound files from the Splash Activity, and is then used in subsequent activities to access music and sound effects. As well as allowing access to music and sound, this class also acts as the media player, and plays music and sound effects from the class.

To use this class, an activity can run a function on an instance of the class, passing parameters such as the volume, and which sound to play. However, on using this, the problem of manually using numbers to access the assets arose. This makes it difficult if assets are changed, or if other sound effects are required. Similar to the Media class as a singleton, a Defines class was created, used not only to define terms for music and sound effects, but as a singleton, it was also used for defining certain terms for options, and even for mathematical terms, such as converting from degrees to radians.

### Separating the Game Activity and Game Surface View

The Game Activity from the interim deadline originally had the surface view integrated into the activity. Although this made it easy to use variables between the two, this also made the code very messy and not very maintainable. I separated this into an activity and a surface view class, however, this could potential break the code, and j-PCT may not actually work when doing this. To avoid changing the main code, while also being able to test a potentially better layout, I created a branch on GitHub, to test separating the two apart.

With a branch on GitHub, I am able to test different things and push them to the repository, without worry of replacing the original code, essentially having a separate repository. Once I am done with this branch, I can then either merge the two, saving the changes I have done with the branch, or it can be deleted, if for example any changes I made did not work, or if it was only for testing.

The use of branches in a project, especially a group project, is very important, as it allows members of a group to work on individual parts of the code, without worry of affecting others or getting affected by others’ changes.

### j-PCT World axis

j-PCT uses its own coordinate axis, which means transforming an object in these coordinates can be difficult. Using Maya to create a scene using objects, transforming in j-PCT would require us to reverse the Y and Z translations, and to reverse the X and Z rotations to set the objects in the right place. Instead of manually reversing the correct numbers, i created a class called TransformFix, which will allow us to pass translation or rotation vectors (as SimpleVector variables) and convert them to the correct format. This can be used as we use j-PCTs transform functions.

(PICTURE OF IN USE FOR TRANSLATION)

### Designing in Maya and loading the scene

For creating a scene, I used Autodesk Maya to position objects and to rotate them to create a scene. The translations and rotations can then be taken to the project, and placed in the desired position and rotation. This is how the Floors classes work, by loading the data in the constructor.

(PICTURE TO SHOW MAYA SCENE AND GAME SCENE, MULTIPLE VIEWPOINTS?)

### Joysticks

To emulate the joysticks you would have on real controllers, I decided to create virtual joysticks, which can be used to move the player, and for the player to look around. I created a Joystick class, which loads textures in 2D, and draws them over the 3D game (similar to the buttons from the interim deadline). These are made up of two parts: the joystick background, which does not move, and the joystick itself, which will move around the joystick background depending on the player’s touch.

Next, I created some functions within the Joystick class, which would limit how far the joystick detection goes, and would detect the distance from a finger to the centre of the joystick, and use this distance to move or rotate the character.

The hard part of this was sensing more than one finger on the screen, as both joysticks need to be usable simultaneously.

### Touch-sensing

When sensing touches in Android Studio, the touch event returns various numbers to symbolise the different types of actions, such as detecting when a finger touches the screen, if it is moving around the screen or if a finger has been released. This does not include if a finger is being pressed down on the same position, causing the joystick to not recognise a stationary finger. This would cause the player to stop moving or turning, even though an input is being passed to the joysticks. To fix this, an integer with different states was used, so it would only turn off when a finger was released. (SHOW PIC?) This worked for both joysticks; however, I was still unable to use more than one finger on the screen. This is because in Android Studio, subsequent fingers are not tracked by the conventional actions explained above, but instead by an action pointer, which gives an index to the subsequent fingers. These subsequent actions also do not have a move action, meaning they could not detect movement from subsequent fingers, and would not update the position of the joystick.

Fig.X Shows the inputs used from the touch screen. With the difficulty of using two joysticks, there were alternatives to consider, such as only using a single joystick to move and rotate the character (similar to the early *Resident Evil* games), or by changing one of the joysticks to a Directional-Pad, which would only detect moving forward, backward, left and right. However, after searching around the topic, I found several topics with roughly the same problem as me, and used a mixture of answers to create the answer for my project. Although it is a bit messy, the fix works properly, and both joysticks can be used properly.

|  |  |
| --- | --- |
| 1st Finger | Subsequent Fingers |
| ACTION\_DOWN | ACTION\_POINTER\_DOWN |
| ACTION\_UP | ACTION\_POINTER\_UP |
| ACTION\_MOVE | No Move for pointers |

Fig.X – Inputs from the touch screen. Only the 1st finger has a move input

Another problem which came up with using the joysticks was switching the ‘first’ finger. If two fingers are on the screen, and the first finger which touched the screen is lifted, the second finger becomes the ‘first’ finger, and will use the first set of inputs. Because of these various problems, the touchEvent function in the GameSurfaceView class is a bit messy, but it has fixed most of the issues surrounding two inputs.

(Reference answers?)

### Buttons

Similar to joysticks, buttons have been implemented as a class, which has a texture which can be displayed on screen. On update, if the screen is pressed, the class will check if the finger is within the bounds of the button. If so, it will return true, and the main code can then do what the button should do. By keeping it generic, it can also be used elsewhere, for example this button class is used for the player to interact with objects, opening the pause menu, and for the buttons in the pause menu.

It has different constructors for different buttons, for example the options and interact button are squares, however the resume and return to menu buttons are rectangles, and need to be scaled to the size of the screen. The way textures work, they must be a size of 2 to the power of n. For example we cannot have a texture be loaded as a 100x100 image. Instead, for the resume and return to menu button, they use a small texture, but the button itself is a different size, so the texture is repeated to fit the difference.

### Text

In the interim deadline, text was drawn as a texture. Although this is fine for a game that only uses a few different words, this project will need to use text at various parts, including potentially subtitles (TALK ABOUT SUBTITLES?). Using j-PCT, I can draw text as multiple textures, where a font or an image of an alphabet is loaded, and separated into multiple images of single characters. Then when drawing text, the text is read character by character, and drawn to the screen.

### Separate levels (combine with scenes?)

To separate the code up better, each floor of the game is created as its own class, which is then used from the GameSurfaceView. As a central hub, it can run the class for the current floor, and read data and move around the map, without affecting the other floors. This is mostly useful for the design of the floor, as each floor is different, and objects on one floor should not appear on other floors.

### Collisions

For collision detection, the game uses a Collision Map class, which can be passed collision areas the player cannot enter. When moving, this class is called, and it checks if we will be colliding with anything in the next frame. If this is true, we will move back to where we were last frame, effectively not moving. On the walls, where a collision is in either X or Y can be checked separately, we can 'slide' along the walls, whereas objects in a room, which have both their X and Y checked at the same time, cannot be 'slid' around.

As the player cannot jump or otherwise move in the Y axis, the collisions can be checked as if they are bounding boxes, checking if a position is colliding an object in one axis, and then the other. In Fig.Y, the floor it is showing in 3D is converted to a 2D collision map, which the player must move around.

(PICTURE TO SHOW SCENE IN GAME, THEN AS A 2D COLLISION MAP (USE MAYA?)) (Fig.Y)

### Keeping UI relative to screen

As explained in earlier parts, certain parts of the game, such as the joysticks or buttons, must be positioned relative to the screen. As there are a wide variety of smart phones (especially with the Android OS) parts of the game must fit within the screen size. For example, the phone I used while creating the game is the Samsung Galaxy Note 3, a relatively large phone with a screen of 1080x1920 pixels. Instead of placing buttons at 960 pixels in X (half of 1920, as the game is played sideways) the button is placed at half the screen width, represented in code using pPoint (SHOW PICS OF CODE) . pPoint has an X and Y variable, which is set the size of the screen. Objects such as buttons and joysticks can then be positioned using this, for example the joysticks are positioned 200x200 pixels away from the bottom left and the bottom right of the screen.

### Inventory and using items

As part of the game, the player can collect certain items to help them progress through the game. The player collects these by pressing the Interact button when close enough to an object, and looking at it. In code, this is simply a Boolean variable, letting us know if the player does or does not have the item. Once collected, a picture of the item will show up on the screen, and can be pressed to then use; for example a key card, which can then be used to open certain doors. To show an item is selected, two versions of the same image of the item exists, one which has a box drawn over it, symbolising it being used (SHOW PIC OF TEXTURES).

### Text Buffer

Text was briefly explained earlier, however, the game makes use of a text buffer, used as a messaging system from the game to the player. This is mainly used when the player is interacting with objects in the game. This works by sending text (from the Surface View) to the Text Buffer class, which then orders the text and draws them to the screen. After a set amount of time, the text is removed from the buffer, and the remaining text is also reordered. This is important for the player, as it allows them to grasp what is happening in terms of the gameplay. Without it, the player would be unable to complete the puzzles.

### Changing floors

The game has a total of five floors (including the Ground floor) which the player can move between. There are also hidden floors, used when the game is completed. To move between these floors, a state machine is used, allowing the player to move between floors properly, and not move to certain floors they should not reach. (DIAGRAM)

### Asset Management

Asset management in Android Studio can be difficult, as certain assets have to be separated, yet some must be kept together. In Android Studio, all images and textures are separated from the other assets, and fall under the category of “Drawable”. Assets under this category are recognised by Android Studio as images for the game, and it uses specific functions relating to “Drawable” which allows us to read and use these images.

For this project, there are a few different types of images used. There are background images for the menu or options, 2D textures for the HUD or inventory, and 3D textures for the objects; however, you cannot create directories within “Drawable” to sort these. There are some workarounds, including not using the “Drawable” directory and using different directories; however this also changes how to load the images, as Android Studio uses its own functions to reach the “Drawable” directory. Instead, the images are named slightly differently to differentiate them.

As objects are loaded with an object name (such as ‘chair’ or ‘table’), these same names are used for the images, allowing the same name to be used when loading the textures. This simplifies creating new objects in the game, as a texture name does not need to be specified. Other images are used as 2D images, and are given the tag ‘img’ before their name, which will stop any conflicting objects trying to load the wrong image as a texture. While images are separated to the “Drawable” directory, other assets are sorted into different directories from the “assets” directory. This includes music, sound and objects.

### Memory Management (IMAGES)

As the platform for this project is a mobile phone, there are many limitations to consider. As well as the physical limitations of a mobile phone – namely the absence of peripherals and controls – there are also the computational limitations. A phone has far less computational power than a desktop computer, meaning memory management, and management of assets, is far more important. At early stages of the project, the size of objects or textures was not a large problem, but as the project progressed, there were signs of the app running slower than usual, and eventually there were crashes pertaining to lack of memory.

With Android Studio, you can run apps, and monitor how they are running, including information on CPU and memory usage, as well as debug errors encountered during testing. By using these monitors, the origin of memory errors could be located. From Fig.A, it was clear that memory usage was too high as soon as the app started. Research into memory management for phone games revealed that the main cause of memory issues were from the assets, specifically the size of textures and objects. On checking the textures, there were a few large textures which could be smaller, as well as file types which used up more memory. The same image as a .bmp file used up more memory than a .png, so they were all normalised to .png, without any loss to quality or transparency.

Large textures were also made smaller where possible. This is a problem often encountered in games creation between programmers and artists. An artist will want to put in as much detail as they can for a texture; however, a programmer will want to keep everything as simple as possible to make it run faster. The textures must also be high enough quality for the player to see, especially if they contain text. The original images for the splash screen and main menu had text, but to further save memory, the text is now a widget, added on top of the images like the buttons. This means the images can be of lower resolution without affecting the text.

All these changes made a big difference, as seen in Fig.B, the memory usage has dropped significantly. The memory usage rises on certain events, such as starting the game, but this is to be expected as objects are loaded in.

### Number pad

As part of the game, a number pad was created, where the player has to input the correct code to continue. This is similar to the pause menu or the elevator, where a window would appear, and stop the player from moving or looking around, as well as give an interface to interact with. A NumPad class was created to do this, which used the Button class to create the buttons for the player to use. Pressing the numbers will also draw text to the screen, showing what the player has pressed.

### Instruction Activity

During a supervisor meeting, where parts of the project was shown, it was pointed out that instructions should be added for the player. Although not originally planned, this is an essential part of any game, as it should not be assumed the player knows how to play or what they should do. To add this, the original activity layout was changed to accommodate an Instruction Activity, which can be accessed from the Menu Activity. This is a simple screenshot of the game, which explains what each button does, and how to use the inventory. (DIAGRAM of new activity layout)

---Not sure to keep below due to changing game---

### Puzzles

The game use a few puzzles as part of the gameplay, requiring the player to solve them to continue with the game and story. When the player interacts with a puzzle, the view changes to the puzzle, (can either continue using current controls, or be able to use 'touch screen' as it were.

The puzzles need to be somewhat short and consise. They arent the main part of the game, more of an add-on or chance to show my skills. They'll be probably logical/ mathematical, probably in the form of moving or using shapes? (Look at 999 or prof layton for inspiration. Research of puzzles? Only if a lot of spare time).

Simplified game

State machine sort of for floors, diagrams, diagrams are good

Explain a bit more about j-PCT

Problems with anything and how to fix and reasoning, JUSTIFICATION

# Testing

### Questionnaires

To judge how good the game is, a questionnaire was made for people to fill in (anonymously) to both get a sense of how good the game is, as well as any bugs which can be fixed. Although testing and bug checking has been done already, there are often small bugs or problems which are easily overlooked by one person. (show questionnaire from appendix?)

The questionnaire has been set questions relating to mobile phone gaming, but mostly has questions concerning the game, specifically certain elements, such as the length of the game (which has been shortened from the original spec), the controls, which is a limitation of mobile phones, and the gameplay and difficulty of the game.

(Refer to actual questionnaires and discuss)

# Conclusion

Critical review of what ive done, what was good and bad.

How could I have improved? Use of different software/ work ethics? Choice of project and shortcomings of the project?

Talk about what I have learned concerning the Horror Vacui stuff, and my findings as such