For the final version, I decided to structure the project similarly to how we structured our mobile game project 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 contains 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 uses a surface view to render and draw objects.

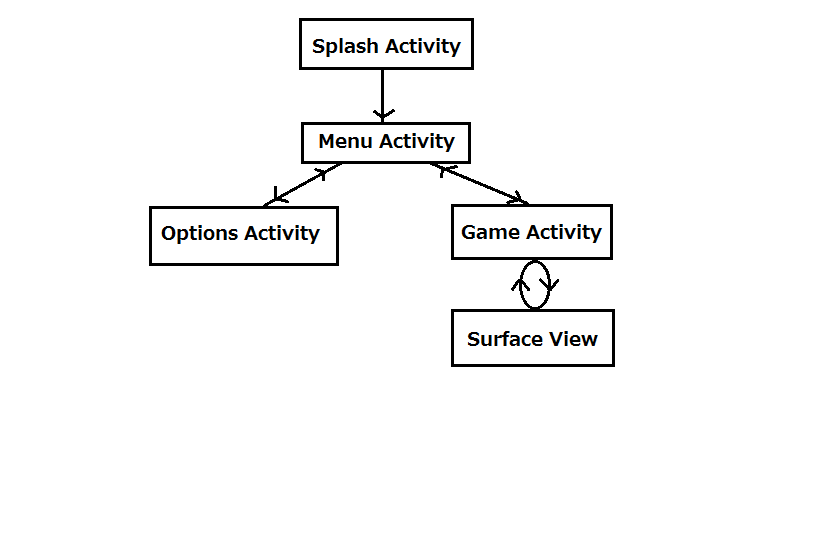
## Activities

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

### 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 to settings to the last used settings. 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. When the game ends, the Surface View will return this to the Game Activity, and it will return to the Menu Activity.

## 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.

## Flow of data through Activities

### Intents

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 is 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 and 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.

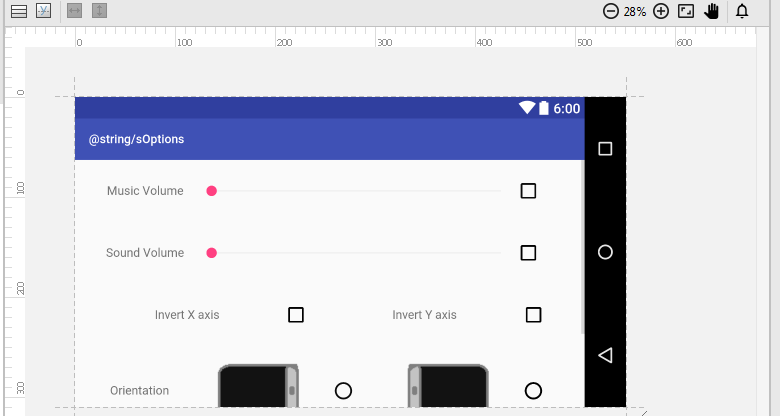


Fig. 1 – The Activity Layout for the Options Activity

In Fig.1 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.

### Singleton Classes

Data can also be passed through a singleton class, 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 is an example of a singleton class, 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.

### 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.

## Write up away from Computer

Write ups for various topics, done away from a computer (such as phone or tablet or laptop) which needs to be checked through.

### 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).

---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