\**section{Introduction}**

This report aims to look at three areas of the accompanying project for the module: looking into the portability of the game that was created in Android Studio by discussing the strategy the group took to ensure the code could was reusable, concise, and maintainable; discussing the option of porting the game to other devices, not necessarily under the Android umbrella, such as the Switch \cite{Nintendo} and the iPhone \cite{apple}; finally talking about the optimisation of the code that was implemented into the game via design patterns to improve the architecture of the software, ending with a discussion into improvements that should be made going forward should the game be launched on the Play Store \cite{playstore}. \\

\**section{Portability of Java}**

This section aims to cover the portability of the Java code used in the project by discussing the approach taken when synthesising the architecture, and how in doing so, it kept the project dynamic in its rendering on any given Android device. \\

\sub**section{Portability}***\label{portability}*

Java has been fundamentally described as a portable language, covering three separate categories; source code, CPU architecture, and OS/GUI. Talking briefly on the source code category, Java is designed with restrictions on its data types which prevent the programmer from defining their length. For example, the integer (int) variable has been limited to 32-bit in the language, without being able to be instantiated to anything more, or less, such as *short int* or *long int* as is achievable in other languages such as C++. This fixed implementation ensures that Java eliminates porting headaches by maintaining a static execution for each of its data types across all platforms, such as Windows, UNIX and Macintosh. \cite{corejava} \\

The CPU architecture portability is determined by the way it is compiled on a given system, by using a virtual machine rather than producing code for a specific CPU family, allowing the code to be flexible. The virtual machine is able to execute the code (referred to as J-code) on a real CPU using this technique. \cite{javaworld} \\

Finally, the OS portability side of Java is given by the language, as similar to the above CPU architecture, the ability to talk to a non-existent operation system. This is solved by Java providing a set of library functions that are able to do this communication to correctly call the GUI of the system it is being instantiated on. \\

\sub**section{Group Strategy}**

When the synthesis of the project was taking place, it was important to guarantee that the game could run on two separate Android devices without running into any complications. For this reason, it was ensured that the programming contained no constant numbers, or variables that were specific to the device being developed on. This eliminated situations such as the placement of enemies, or other game assets, that were static to a section of the screen on both a larger and smaller device, allowing for a more dynamic approach which rendered placement and scale in relation to the device. \\

\begin{figure}

\centering

\includegraphics[width=\textwidth]{android}

\caption{Android Studio’s account of the Operating System’s most used versions.}

\label{fig:androidstudio}

\end{figure}

The game was also implemented with the ability to switch the control schemes between accelerometer and standard touch to allow for older devices without the functionality of the accelerometer to still be able to play the game effectively. Likewise, the project was setup to run on older versions of Android, with a very large percentage of systems still utilising the Jelly Bean operating system, according to Android Studio, the evidence of which can be seen in Figure \ref{fig:androidstudio}. In addition to this, Jelly Bean (5\% of all devices) and many of its successors are still being used on a large section of the Android ecosystem, with the newest, Oreo, having a relatively small install base (1.1\% of all devices) presently. \cite{fossbytes} \\

\**section{Transferability to Another Platform}**

In Section \ref{portability}, it was determined that the Java language is very easy to port to other devices that utilise Java as their primary language of choice. When porting to other devices, there are a few obstacles to overcome with other devices preferring Objective-C or even C++ as their language of choice. This section aims to cover these obstacles and how to overcome them. \\

\sub**section{Achievable}**

To answer the question of a direct port to another platform, in the event of a literal copy and paste to the appropriate target IDE, the answer is that it is impossible to do so. For example, the Apple iPhone uses Objective-C for its structure, and as such, the game would not be compatible with this architecture, and would not be able to execute on this platform. \\

There are methods to solving this issue, with a number of tools available to port applications to other devices by making intelligent decisions to automatically re-write the code to another language, or to have the programming painstakingly re-write the entire code base to the platform of choice. \\

\subsub**section{iPhone and iOS}**

Porting the game to the iOS ecosystem, as mentioned in the previous section, is impossible as standard with Apple choosing to develop their software using Objective-C. There is a work around that Google themselves have created, a command-line tool called J2ObjC \cite{j2objc} that is designed to specifically translate Java code directly to Objective-C, but should only be used for the business logic of the code, and not for the user interface. In all cases, the UI should be programmed manually for the target application, as there is no way to perfectly preserve the semantics of this type of code. \\

Another way to port Android applications to the iOS ecosystem is a plug-in called Multi-OS \cite{multios}, which can be used directly with Android Studio. The programmer must first, in a similar way to J2ObjC, design the user interface for the application in Apple’s Xcode \cite{xcode} IDE before binding it to the Java NatJ runtime libraries. Android Studio can then use its Intellisense capability to bind the action handlers to the UI elements for the iOS platform. \\

Using this methodology, it is possible to port the game to iOS, but it isn’t an easy fix, with all decisions to port the game having to require extra work from the programmer to ensure it runs as desired. \\

\subsub**section{PlayStation Vita}**

Porting to the PlayStation Vita \cite{psvita} system takes a slightly different approach, as the Vita uses C++ as its primary language, and is therefore not compatible with the tools used for iOS. Instead, the project would have to be slightly re-wrote for C\# before it could then be made to work on the Vita platform using Sony’s Unity for PlayStation Mobile utility. Whilst this is hardly an ideal situation, C\# has been described as being syntactically similar to Java \cite{csharp}, with very minor differences being present, which would arguably make the conversion more bearable than it would be to transcribe it to C++ or Objective-C. The Unity utility would then port the project to the Vita where it would eventually execute using the native C++ architecture. \\

For this reason, it is a better decision to choose an IDE that is capable of distributing the project to a number of systems initially, as opposed to developing for an Android specific IDE and going through the headache of porting to other languages as an afterthought. Unity, for example, is capable of running its projects on a number of platforms as standard. If, however, the programmer intends to run the game on Android devices only, it is a far better decision to program in the native language, in this case Java, as it should ensure the project runs optimally on the given platform. \\

\**section{Optimisation}**

When creating the project, there were occurrences when optimisation had to be applied to keep the project running smoothly on the device. The first instance of this occurred from firing bullets from the player ship; these objects were created dynamically, then move by some vector across the screen indefinitely, remaining in the system until it was closed. This would eventually lead to the system running slower as the game progressed. To alleviate this issue, there would have to be a system developed that could process the collisions of all objects in the system, which would allow for the deletion of those objects so they could have their memory freed up for other parts of the program to use. \\

There was also an issue in that collision detection between objects had to have a lot of $else if$ statements present to perceive the appropriate response depending on the object it was colliding with. This led to a lot of repeated code, which is obviously inefficient. For example, the player ship would have to detect a collision with the three different types of enemy, in addition to the missile drops that could spawn when an enemy has been defeated, and this would have to be done every frame. In addition to this, there were separate classes instantiated for each game object; i.e., a player class, and an enemy class. These objects had no way of communicating with each other, as their types were different. \\

\sub**section{Improvement Methodology}**

To tackle both the memory and communication of the game objects, there were a number of design patterns used. \\

The first method that was implemented was an abstract class, GameObject, of which all objects in the game would inherit from. This base class included basic functionality that all objects in the game should have, such as a bitmap assigned to it for rendering the object, two integers which represented the x and y coordinates of the object, a hitbox, a Boolean to determine whether or not the object is active in the game, and a method for how the object handles a collision. In addition to this, functions could take a GameObject as a parameter, and all objects inherited from this class could be passed through it, making the system polymorphic in nature, as the method would interpret which object it was managing at run time. \cite{adobe} \\

To alleviate the problem with memory, another design pattern was implemented: the object manager. This class handled all of the objects in the game, from deletion of the object, to checking collisions between the objects in a meaningful way, and was also instantiated as a $singleton$ class, meaning there could only be one implementation of the object manager. This was implemented by creating a list in the class, $m\_allObjectsList$, which stored every GameObject that was created in the game. This list could then use a nested loop to check each object against another in the loop to see if they had collided, and if they had, run the $ProcessCollision$ method for both objects. The object factory could also alleviate the issue that was present with the player endlessly firing bullets by deleting them from both the list, and the system, after they had been determined to be expired. Another way in which the object factory kept the game running smoothly was by looping through every object in the game and checking the Boolean associated with it, $m\_active$, to see whether it was set to true or false. If this Boolean was set to false, the object manager would remove it from the $m\_allObjectsList$ and delete it safely from the system, freeing the memory up for other objects to be instantiated. \\

\sub**section{Recommendations}**

One recommendation to improve the architecture of the project is to enhance the object manager class so that it handles the construction of objects, and the amount of memory given to each type of object, in addition to their management by following the object factory design pattern. \cite{zhang} Object factories are ideal for platforms that are memory restricted by putting a cap into the number of objects that can be created at a given time. For example, there could only be ten enemies in the game at once, and any new enemies would only be created once the player removes one from the game. In addition to this, the possibility of memory fragmentation would be completely nullified, as the section of memory assigned to an object type would not accept other object types, thus each segment would be a perfect match. \\

\begin{figure}

\centering

\includegraphics{fragmentation1}

\caption{How memory fragmentation looks in a normal system without an Object Factory. \cite{fredrik}}

\label{fig:fragmentation}

\end{figure}

Figure \ref{fig: fragmentation} shows how memory behaves currently in the game, with object deletion leaving gaps between the memory currently in use. This eventually leads to fragmentation when an object cannot be instantiated within one of the resulting gaps. \\

\begin{figure}

\centering

\includegraphics[width=\textwidth]{ memoryFactory}

\caption{Memory efficiency using an Object Factory. \cite{nahuel}}

\label{fig:fragmentation2}

\end{figure}

In Figure \ref{fig: fragmentation2}, we can see a chunk of memory that would be allocated in a system using the Object Factory class design pattern. In this case, memory fragmentation cannot possibly occur, with each object being the same size, and therefore, able to fit into each chunk in the segmented memory. \\