BSc (Hons) Computer Science

Comparing Game Maker and Java as a method of creation

Matthew Pool
B00278639

23rd March 2018

Supervisor: Chris Armstrong

Declaration

This dissertation is submitted in partial fulfillment of the requirements for the degree of [Programme of Study] (Honours) in the University of the West of Scotland.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself. Following normal academic conventions, I have made due acknowledgement to the work of others.

Name:
(In Capitals)

Signature:

Date:

Form to accompany Dissertation

Surname: Pool
First Name: Matthew **Initials: MWP**
Borrower ID Number: B00278639
Course Code: COMP10034
Course Description: BSc (Hons) Computer Science
Project Supervisor: Chris Armstrong
Dissertation Title: Comparing Game Maker and Java as a method of creation
Session: 2017/2018

Project title: Comparing Game Maker and Java as a method of creation.

Student: Matthew Pool Banner ID: B00278639

Supervisor: Chris Armstrong

Moderator: Rebecca Redden

Outline of the project:

The project shall be used to investigate the difference between game development software and Java using the creation of a game. With the outcome showing which is better for the developer and which is better for the games performance. In doing so will this will hopefully provide support for any upcoming game developers and in the education of users.

This will be completed using a common design which will be constructed before the main development is started. This will ensure that both versions will follow the same rules and specifications.

The results will hopefully show which method provided the most improvement and comprehension with regards to programming ability and problem solving. Also, the final deliverable will be able to compare as to which method showed a greater efficacy and performance. Which may also come into the recommendation.

A passable Project will:

- The deliverables should be completed and compared fairly.
- A report based on the results of the testing and development which discusses the comparison of the two methods.
- The project will be deeply researched to ensure the project is fairly conducted.
- The game design will be conducted thoroughly and clearly using development techniques.

A first-class project will:

- Testing should be conducted fairly and properly to ensure the same conditions are met for each test conducted.
- The conclusion should contain an analysis based on the development and the results and testing.
- As well as the deliverables, the report should contain detailed analysis of the process of development and implementation. When comparison is made it should be specific to the nature of the project with clear identification and analysis of the results.

Reading List: Killer game programming in Java, The game makers apprentice,

Resources Required: Computer using a Java IDE such as IntelliJ or eclipse, Game Maker 8.1 or Studio.

Marking Scheme:

	Marks
Introduction	5
Literature Review	15
Game Design	10
Development	35
Testing	10
Conclusion	15
Critical Self-Appraisal	10

Signed:

Student: Matthew Pool Supervisor: Chris Armstrong Moderator: Rebecca Redden Year Leader:

Table of Contents

Acknowledgements

I would like to give thanks to both my supervisor and moderator, Chris Armstrong and Rebecca Redden, who have answered many questions and supplied much needed advice. I would also like to thank miss C.Swailes for her tireless efforts to aid in proof reading and for her morale support.

Abstract

The growth of the game industry has led to new companies being created and more interest for those wanting to get involved. This study looks to compare creation methods of games and whether to use game engines or programming languages. Game Maker and Java were compared against each other through the creation of the same game in each method, which was then assessed by factors relating to the development process and by performance-based result. It was found that Game Maker produced a more efficient game in terms of processing power for both the CPU and GPU, where Java produced a game that required less memory to operate and store. Developing using Game Maker was found to be a much more forgiving and efficient process while Java's development was much more complex. Therefore, Game Maker was found to be the greater of the two development methods.

Chapter 1 – Introduction

1. Introduction

The proposed study aims to compare game making software with more conventional programming languages by creating a simple 2D game. Several factors will be applied to measure the effectiveness and benefits of each development method. The area of study was chosen as there has been a rising amount of interest in the UK games market with more people and companies becoming involved every year. The number of registered game development companies has increased by 68% between 2010 and 2016 (UK VIDEO GAME FACT SHEET 2017). Therefore, this presents an opportunity for new developers to take advantage of the growing market and begin developing themselves, meaning they will require guidance as to how to begin game creation.

With many tools and platforms available for game design, it is essential for developers to select the appropriate one. There are hundreds of programming languages and many forms of game development software available. Java was selected as the programming language to examine, as it was the most popular programming language as of March 2017, with a rating of 16.4% on the TIOBE index (TIOBE.com, 2017). This popularity demonstrates how widely searched for java is as a programming language and may predict its continued success and growing popularity in the future. Java has also been incorporated into teaching to aid learning of programming concepts within Computer Science, even though it was not specifically designed to do so (Mannila, de Raadt 2006). Java also has the capability to make any game and can be combined with pre-made libraries specifically for game development such as LibGDX (Zechner, 2013).

Game Maker was selected as the game development software as it is specifically designed for 2D game development and has all the necessary tools. These include a drag and drop system as well as a language specifically created for the software which offers more control (Yoyo Games, 2017). Students within the Computing Science courses at the UWS are likely to have encountered Game Maker previously. With this setting, Game Maker has shown to have positive effects on student's attitudes towards the course and their lecturers (Doman, Sleigh and Garrison, 2015). Game Maker's design processes act as stepping stones to more

complex programming concepts. This is especially true with its Game Maker Language (GML) which allows the learner to learn Object Orientated concepts (Hoganson, 2010).

When comparing these different methods of game creation, it is necessary to select an aesthetically clean, easy game design to replicate. The 2D side scrolling game genre are among the easiest games to make. According to a poll conducted by escapist magazine, platformers are said to be the easiest genre in which to create a successful game, with 38.1% of respondents supporting this idea (The Escapist, 2017). The side-scroller exists within the platforming genre as it is in a 2D format on a single screen. However, the side-scroller proposed within this study will also exist in an infinite loop, a quality which is not universally associated with the platforming genre. This concept will provide a simple basis for game design in both Java and Game Maker.

Chapter 2 - Literature Review

2.1 - The game industry

Technology advances with the development of greater hardware and as a result so does the game industry. With better hardware new game engines with higher capabilities can be created and computers that can process data even faster. These developments also give rise to higher level programming languages which are more powerful than ever before. The wide variety of game engines and programming languages have the potential to make becoming a games developer more accessible. However, this choice also presents a challenge to novices who may be unsure whether they should invest time in learning a programming language such as Java, or alternatively use pre-made 3rd party game engines such as Game Maker.

The game industry has developed considerably from the 1990's where classic games such as DOOM and Quake were first developed using ID software's "DOOM engine" which was later released as one of the first 3D rendering game engines (Lowood, 2014) (Guins and Lowood, 2016). The video game market reached world sales of $20.8 billion in 1994 (Lange, 1996) and began to expand substantially, reaching a total value of $91 billion in 2016. This demonstrates the enormous growth of the game industry worldwide in the space of 22 years. The UK game industry is currently the 6th largest in the world, being associated with 2044 active game companies and obtaining a total revenue of £3.8 billion in 2016 (UK VIDEO GAMES FACT SHEET, 2017). Given that the UK game industry's revenue was £1.2 billion in 1994 (excluding inflation), the UK market has expanded enormously, much like the worldwide sales. The steady growth of the industry in recent years suggests that it will continue to expand, and so a market exists for the future development of games. As with the financial side of the game industry the development aspect has also expanded along with the platforms in which the games can been played on. Development has advanced to where there are development environments for every niche game genre. There are general purpose third party software such as Game Maker or Unity3D or more specific environments such as Pico-8 for smaller retro style games (Lexaloffle.com, 2017).

Companies and developers can create their own game engines in house or program games from scratch with the language of their choice depending on the demands of the game.

2.2 - The game engine

A game engine is a pre-written framework that provides developers with the necessary tools to build one or many games. The game engine can be divided into its core components, the audio engine, physics engine, rendering engine, the AI engine and input systems. Each component controls specific parts of the game which is essential for its functionality.

The Audio engine controls the sound effects which are employed. This is very important as it allows for enhanced player immersion and represents a critical aspect of the game. The Rendering engine controls what the player sees by calculating where the pixels must be displayed and then outputs this. The renderer uses application programmable interfaces (API) to make the process more efficient and less taxing on the Graphical processing unit (GPU). The Physics engine controls and provides systems critical to any game such as collision detection. This calculates whether an enemy has been destroyed or the player is in mid-air and gravity needs to be applied, physics play a crucial role in any game engine. AI is essential in a game engine as it controls the logic of the world and all non-playable characters (NPC). The input system is used to obtain, control and interoperate the players commands as they occur and relay this information to the physics and the rendering engine to show the output of the selected input (Kalderon, 2011) (Nandy and Chanda, 2016). Game engines have been described as belonging to three categories by Ward (2008), including: roll-your-own, mostly-ready and point-and-click. Roll-your-own at the lowest level involves the developer using APIs to essentially create essentially their own game engines which will require a high level of programming experience whereas at the other end of the scale the point-and-click game engines provide all features necessary for game development and require little to no prior experience to get started. Game Maker belongs to the point-and-click category this is due to its built-in features which cover every need for 2D game development.

Game Maker is a product of YoYo games which was founded in Dundee in 2006. It is a game engine specifically designed for 2D game creation and provides both its own language (known as the GML) and a drag and drop style creation method. The drag and drop method includes various predetermined events that can be applied to objects and to the rooms in which the games take place, such as when the collision event occurs between certain objects that will then trigger a sequence which can be written in code or in the drag and drop system. When combining these systems, creation becomes intuitive and easy as the flow of events are clearly defined (Yoyo Games, 2017) (Habgood and Overmars, 2013). Some of the most successful productions which utilised Game Maker include the indie games "Hotline Miami" and "Hyper light drifter" (Yoyo Games, 2017) (Fingas, 2017) (Batchelor, 2013). The creation method when using Game Maker will be very important within the proposed study, as it will become a focus for showing the benefits of the software. Game Maker has simple components at its disposal and uses rooms to act as the level or environment of the game in which actions take place and objects are stored. Objects are key within Game Maker from the player character to the enemies and ground the player moves on, anything that has functionality is generally an object. Sprites and background represent the colour in Game Maker, providing a visual representation to the user. Sprites are applied to objects while the backgrounds can be applied to rooms. With these four basic components games creation can begin (Habgood and Overmars, 2013).

2.3 - Programming for Game Development

Game development though programming alone can be a long and arduous task without the game engine carrying out procedures such as sorting and calculating the graphics and physics. When developing from the ground up programmers must create all the same processes that are contained within game engines, showing that it can take significantly longer to develop with programming alone. In the past high-level programming languages were rarely in game development used as they were too intensive on memory and computing power. Instead low-level languages were used as they provide less abstraction from the hardware and could bypass the performance issues. Languages such as Assembly have a closer relationship to the actual hardware of the computer to gain the most out of

the machinery without compromising memory and processing power (Playitagainproject.org, 2013) (Comer, 2017). One of the well-known games written in Assembly is Rollercoaster Tycoon which was released in 1999 (Sawer, 2005). As the memory and power constraints were reduced, high level programming languages became more common as they allowed for a greater level of abstraction from the hardware. This in turn made them easier to comprehend and learn. The high-level languages made flow control easier with simple loops and could be applied across many machines while the like of Assembly would be written for specific hardware (Techwalla, n.d.).

The process of creating a game through programming alone is done through the creation of a few simple but critical systems (Figure 1). These systems include the game loop which controls the flow of the game, the speed of the animation's refresh rate and frequency of calculations within the game. The refresh rate is commonly set to 60 frames per second (FPS) which represents the number of calculation repetitions the game makes every second. The game loop contains the processes that calculate graphical displays, which is also known as the *renderer*. The main loop also contains a method known as the *game state* which obtains input from the player and processes what the game should look like for the next frame by relaying that information to the renderer. This is a basic version of the game loop which is used when dealing with multi-threaded applications to ensure that the loop is exactly on the right time. A time handling method exists which will calculate the time taken for each iteration of the loop. If the loop is running fast the time handler will force the loop to pause for the correct amount of time to prevent the FPS from deviating (Davison, 2005) (Madhav, 2013).

```
while(run){
        Gamestate();
        Renderer();
        sleep();
    }
}
```

Figure 1: Basic game loop

Java is not the most well-known game development language and there are many objections to its use in game programming. Java has been considered too slow for game programming and has demonstrated memory leaks. These represent the main concerns when considering using Java as a game programming language, yet, according to Davison (2005) these are particularly common misconceptions about the language. Davison suggested that memory leaks occur as a result of bad programming practice, arising when

objects are continually created without being correctly dereferenced, therefore the Java garbage collector will not destroy the objects and the memory will become saturated. In reference to Java being "too slow" for game programming, this usually means that people are comparing Java to C++. However, with its recent updates, Java is closing the performance gap with C++. Recently Java has demonstrated several advantages over native C++ such as a reduced size of the executable and the resource size as well as the garbage collector which assists in the prevention of memory leaks that a C++ programmer could forget about (Mangione, 1998), (Davison, 2005). These findings suggest that java is at no disadvantage when being compared to its supposed biggest competitor for the creation of games. Java's suitability as a beginner's language has also been evaluated and compared against that of C++. It was suggested that Java possessed concepts which were easier to comprehend, especially regarding memory management. It was also found that performing debugging of programs was considerably easier with Java which allowed the learners to be more productive with this language (Irimia, 2001). Some of the most popular games which utilise the Java language include Minecraft which has been sold to Microsoft by its creator for £2.5 billion (Sheffield, 2015).

2.4 - Comparison of game engines and programming languages

Comparing different development methods in computing is common, yet, in the field of game development few studies exist, and little content is available with regards to comparing a game engine against a programming language. The lack of information available on the comparison of game engines against programming languages increases the value of the present study yet it will require greater extrapolation from related topics of research. For this section discussion of comparisons between programming languages will be discussed as well as comparisons between game engines to determine better approaches for the present study. The steps involved in comparing each development method are an important factor to consider and will underpin how the comparisons will be made.
The most important part of any comparative study is the criteria which will be used to measure the performance of the games being tested. Comparison of game engines is

frequently related to their features as they are largely very well documented in terms of their capabilities. Comparing game engines in this way often leads to a result which fails to suggest that any one is superior over the others as each method is more efficient when used for the specific purpose it was designed for (Christopoulou and Xinogalos, 2017). Developing a more specific criterion and choosing a niche use for development methods will aid in reducing the chance that there is no definite answer at the end of the comparison. To achieve this specificity, a custom set of factors can be created to improve the selection process (Pavkov, Franković, and Hoić-Božić, 2017). The type of criteria chosen must reflect the type of study and considering that analysis will be based on a performance comparison between each method performance-based criteria will be selected. The comparison of programming languages often uses this form of comparison where the efficiency of the code is measured, along with the creation of an application (Parveen and Fatima, 2016). As the present study is comparing a programming language to a game engine, the construction of specific criteria on a performance-based study design will be incorporated.

Chapter 3 – Design

3.1 - Game design

The importance of the design process to the results of this study is critical. The design process will help ensure that a fair measure of the different development processes is obtained by ensuring that a game is created which is most suitable for game maker, java and the developer. The game design process must also factor in that the game is achievable and can be created within a limited amount of time by a novice developer, while still being practical for testing. The conclusion of the game's design was that an infinite style runner would be created which would become increasingly more difficult as the player progressed through the game. Planning of The Game's the game's design began with consideration into which game types would be best suited to analysing how Java and Game maker differ in terms of performance. Performance-based results required continuous calculations to be running and using processing power. This ruled out the possibility of developing "point and click" games as well as other static games, as they often do not require a game loop which will update the game every time the loop iterates.

The choice between 3D and 2D games was also an important consideration as although many different game engines provide 3D support, creating a 3D game in java will require in-depth knowledge of rendering systems and more complex algorithms. The fact that 3D rendering is not built into Java would mean that the development time would escalate greatly, making the present study impossible to complete. This issue narrowed the choice of game genre to a 2D game design which can be produced and completed in less time and requires less knowledge on the developer side. The use of 3D rendering would likely provide better results due to 3D games requiring more processing power and could amplify any results identified in the comparison of a 2D game (Blog.rogach.org, 2015).

The limited amount of time available for design and development of the game also provided new restrictions regarding the genre of game. Therefore, the game could not consist of complex content or any features that would render the game impossible to create in the time frame provided. Thus, genres such as Role-playing games (RPG's) or games consisting of multiple levels could not be created. With the conditions of time restriction in mind, the

arcade style game which continue infinitely fit the needs of the project perfectly, where games progressively become more challenging until the player is defeated. The infinite style can use random chance to spawn new obstacles, so these do not have to be placed before hand and there is no chance for them to run out. The game can accelerate the spawn rate or the speed of the game to make the game gradually more challenging and eliminate the need for any form of significant progression of a storyline.

With the amount of content, the dimensions that the game is played in and the expected results of the game in mind, it is narrowed down to a 2D game which can be played infinitely whilst becoming more challenging and can be measured in terms of performance. The choice of games that can be made within the bounds of these restrictions varies greatly but due to the simplicity of the infinite runner (which is already a genre on its own) has been chosen due to the developer's personal preference.

The game's design was approached in a way that would ensure that the final developments of each game would be the same or at least as similar as possible. The establishment of a set of guidelines to govern the production of the game provided stability. These guidelines aided in the control of the game creation process by providing the developer with a basis for the construction of each game and the games limitations.

The guidelines are as follows:

In-game guidelines

- The game will run at 60 frames per second.
- The gravity force shall be standardised to 9 pixels per loop.
- The maximum jump height will not exceed 2.5 times the height of the player sprite.
- The world will be on an infinite loop.
- Auto generation will control the landscape and obstacles while ensuring that it remains possible to jump onto all new platforms.

Development guidelines

- Window size will be restricted to 600 pixels height and 800 pixels width.
- All measures will be calculated in pixels.
- A similar style of object orientated programming will be employed following the class diagram where possible.

- The same assets will be used in each game including player sprites, obstacles, objects and backgrounds.

The simplicity of the design and guidelines benefit the study and the accuracy to which they can be followed to create a replicated game. The game design process after these guidelines were established focused on the type of game play that would be employed. This game play included how the game would look and feel, whether spites would be animated, the type of challenges there would be to overcome and the reward the game would provide, be it a high score or another achievement within the game. The game will consist of basic components that will make up the core of the game. These components include the player character, the ground and the obstacles the player must get past to survive. The player will control a sprite on screen with the W A D keys to move and jump. The game will randomly generate obstacles for the player to avoid starting with obstacles that stay with the ground while projectiles will be spawned as the player progresses further. The players score will be recorded as a distance that will be displayed in the top right corner.
The final game design can be seen in figure 3 where it depicts the player, the cyan rectangle, jumping over a moving obstacle.

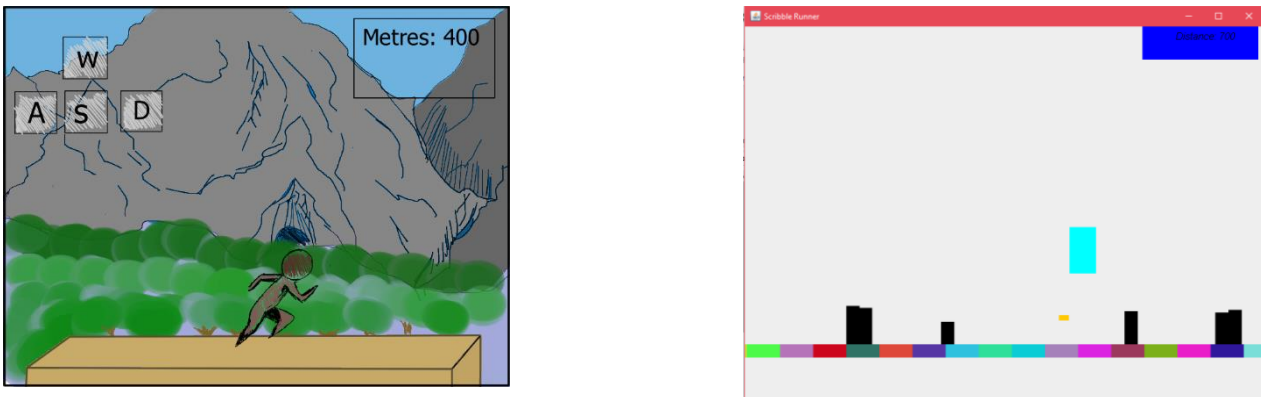


Figure 2: Original concept for the Scribble runner eted version of the game

Chapter 4 - Game development

4.1 - Development Approach

The development process has been completed using an Agile framework to make the process more fluid especially as the development will also be a learning process. This fluidity is achieved with the iterative style of progress employed by Agile. This will be especially important for correcting mistakes in the development without affecting the schedule. In recent years the Agile method has advanced and has been implemented in more projects in software development. The development of Agile has also led to different branches of this framework such as the Scrum variation, which has been adopted in the present development project. The Agile Scrum framework achieves its flexibility as its structure encourages developers to use cycles of development, much like the regular Agile method. Scrum includes short "sprints" of development that occur recursively until the client is satisfied with the outcome. When each sprint is started it focuses on one feature or aspect of the development at a time and will continuously update this feature until the developer is satisfied. The Scrum framework is predominately used by teams of developers. However, the present study has been conducted by one researcher, yet the Scrum concepts and approach will still be utilized. The Scrum management style employed will focus greatly on daily meetings where the plans and research will be planned before a sprint occurs. Sprints usually occur over 2-4-week periods, yet in the present study they were shortened to a couple of days (Cohn, 2017). These adaptations will be unconventional in comparison to the traditional Scrum method, but the project is required to be completed by a singular developer and the scrum management framework was the most appropriate for both the developer and the project.

The development took place over several weeks by first completing the Java version and then the game maker version of the game. With the assistance of planning aids such as the class diagram and the Gantt chart, time lines were established for the creation process. The process of development was separated into different sections that would each give functionality to the game. The sections were then used as the basis of the Scrum sprints that would construct the development process. The different functionality began with creating a screen that updates at 60 FPS, then populating the screen with moving objects for the

games floor, adding a player character that responds to the rules of the game, setting up collisions for the player and the floor, creating player input, inserting obstacles for the player to collide with, setting up collision conditions for the player and obstacles then finally introduction a measure for the players progress and all game ending conditions.

4.2 - Java

Java is an Object-Orientated programming language which means it harnesses several key features, that other types of programming languages do not have, including encapsulation, polymorphism and inheritance. The games functionality was created through the construction of classes which are collections of variables and methods. There are generally multiple classes that link together to form a program or application. The structure of the game and its classes can be seen in the class diagram in figure 4 where it depicts the dependency of the games classes on each other and how they interact. The different classes all serve a different purpose that is critical to the playing and running of the game. The development of these classes however, occurred in different stages as functionality was implemented. The entire Java game can be viewed in Appendix A where It has been separated into the different classes.

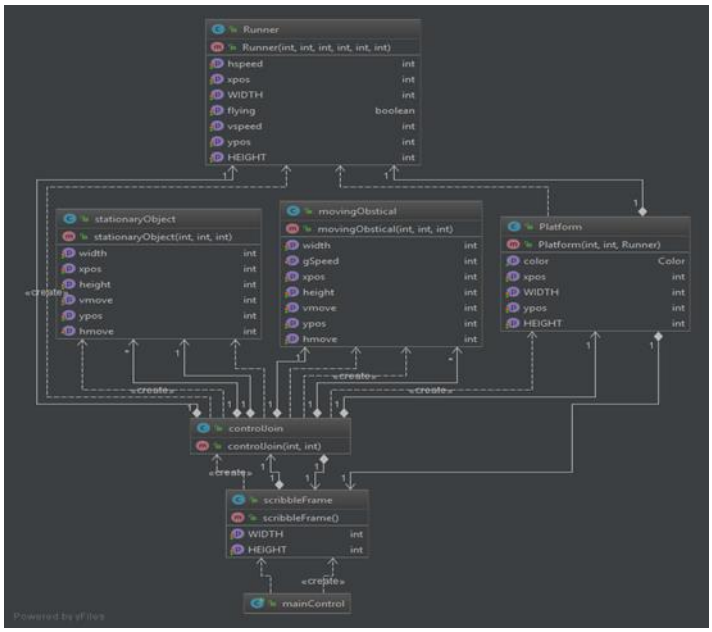


Figure 4: Final Java class diagram depicting the connection, the constructors and the properties of the classes.

4.2.1 - Creating the Window and Game loop

The first step in the creation process was to create a space for the game to take place, this was done by initiating and creating a JFrame, known as the "scribbleFrame", and then adding a JPanel, known as the "controlJoin", within it. The JFrame is essentially the window that can be seen when applications open and the JPanel is added to it as it is where content can be added and created within. The JPanel class houses most of the content for the game including the game loop which is the core of the game. The set-up of this initial stage has been adapted from the "killer game programming in java" book that gives guidance for the complexity of the game loop.

The game works based on one main thread which the game runs on this thread is what allows the game to run and for the game loop to work. Upon first implantation of the loop it was found that there were issues with the panel trying to start the thread before the panel had fully been created and some variables that were trying to be used had not been initialized yet. To stop the problem of the thread trying to start too soon, a new sequence was created with the "addNotify" method which only allows the thread to start once the JFrame and JPanel are visible and initialized. The "addNotify" method can be seen in figure 5 displaying how It connect to its counterpart "startThread" to initiate the thread and subsequently the gameLoop.

```
public void addNotify(){
    super.addNotify();
    startThread();
}


public void startThread(){
  if(!running || gameControl ==null) {
      gameControl = new Thread( target: this);
      gameControl.start();
  }
}
```

Figure 5: The addnotify() and starthread() methods can be seen and how they are structured

Once started the thread activates the Run method which in turn houses the game loop, the game loop contains a basic sequence of events that power the entire game. The game loop contains the update method, the render method and the "activePaint" method. The update method controls all the changing of data as the game runs based on player input and other calculations, the render method draws all these changes onto an image that is then

displayed by the “activePaint” method and this repeats until the game ends. The game loop keeps the game running at a specific FPS. The FPS is calculated by taking a measurement in Nano seconds from the computers CPU before and after the game loop iterates, the difference Is then measured between the two and this calculates how much time has passed to complete all actions within the game. The time taken to complete all tasks is then subtracted from the period of time that would result in a perfect 60 FPS loop, the resultant time will then be used to send the thread to sleep to gain a perfect loop that will now operate at 60 FPS, this process can be seen in figure 6. The process of working out how much sleep is needed as It regulate the game from running too fast or too slow. The game running too fast is the main issue when regarding the game loop, although the potential to run too slow was also addressed by setting a method that will allow the game to catch up by calling the “update” method alone instead of the “render” method to lessen the load on the computer. Testing of the game loop was done throughout the implementation to ensure it ran a correct and constant speed and was also tested later once painting to the screen had been completed.

```
beforeTime = System.nanoTime();
    while(loop){
        update();
        gameRender();
        activePaint();
    afterTime = System.nanoTime();
    //conversion from nano to milli then to the second time
    timeDiff = (afterTime - beforeTime);
    sleepTime = (period - timeDiff)-overSleepTime;//oversleeptime to cat
    if(sleepTime>0) {
        try {
            Thread.sleep( millis: sleepTime / 1000000L);
        } catch (InterruptedException e) {
            overSleepTime = (System.nanoTime() - afterTime) - sleepTime;
        }
    }
```

Figure 6: This demonstrates the main components of the game loop and how it calculates how much sleep is required

4.2.2 - Creating objects

The first objects to be created in the game were a set of platforms that would become the ground, the platforms were first created as a single block floating across the screen and then as an array of 17. To begin the implementation the Platform class was created and then initiated in the JPanel class where basic values were set for the width, height. To see any

changes in the platform the update method had be used to calculate the change in position of the platform, so a move method was created in the platform object to set a new x and y position of the platform for every iteration of the game loop. The render method was then set up to create an image and a graphics object where a rectangle with the height, width, X position and Y position of the platform would be added. The "activePaint" method in the game loop was then coded to use the graphics object to paint the platform to the window visible to the player. To create multiple platforms a loop was set up to initiate 17 platforms and set there X positions to be there width apart from each other to give an indication that the floor was continuous but because of this, the movement of the platforms was disguised by their uniformity so each platform was assigned a random colour upon creation to make sure it was obvious that the platforms were moving. The platforms were then implemented in the form of a cycle where once the leading platform had fully left the screen it would then have its new X position set to be just off the screen on the opposite side of the window . To further improve the platforms a draw method was added to the platform class to make the render method less cluttered and easier to comprehend.

4.2.3 - The Player

Adding the player character

The sprint to incorporate the player object consisted of creating class "runner" with variables for player positioning, the width and height, the players speed and external forces, such as gravity. During the first iterations of the player implementation both drawing, and the moving of the player were completely within the JPanel class, but by moving the draw and render methods into the "runner" class this neatened the code and made retrieving data easier as the move and draw methods did not have to reference the player object but could directly access variables needed. The player object was added to the JPanel class and then initiated with a stating positions as well as its current speed. The speed of the player without input was set to -1 in a horizontal direction in order to move with the platforms and to give the appearance that the player is stationary. In the Move method for the player the change in X and Y position depended on what values were assigned to the horizontal and vertical speeds. These changes in position would be calculated every time the update method was called by setting the new X or Y position by adding the Horizontal or vertical

speed to the previous X or Y position. The draw method would then then be called by the render method to redraw the new position of the player.

Collision Detection

The next step for creating the player character was allowing the player to move according to gravity and stop when hitting a solid surface. The "currentCollide" method is used to check for the players position against every platform on the screen, its main functionality can be seen in figure 7. When the player is touching any of these platforms a true value is returned which tells the move method to no longer apply gravity. The "currentCollide" method determines the players contact by first calculating whether the X positions of the player character is within any of the platforms X positions, while at the same time making sure that the Y positions are also aligned and if these values both return true then the player is on the ground and thus a true value will be returned.

```
for(int i = 0; i<pTest.length;i++){
    int currplatXpos = pTest[i].getXpos();
    int currplatYpos = pTest[i].getYpos();
    int platWidth = currplatXpos + pTest[i].getWIDTH();
     if(playYpos >= currplatYpos){
            //player iss on the ground as you increase further down you go
         if(playWidth >= currplatXpos && playXpos <= platWidth){
            // if this returns true then they are on the ground.
            // need to reset player pos
            player.setYpos(pTest[i].getYpos()-player.getHEIGHT());
            return true;
        }
```

Figure 7: The onTheGround method that is used by currentCollide to check for collision

Player Input

To get horizonal and vertical speeds player input also had to be added, this was implemented by adding an "actionListener" sub class within the JPanel. To use the "actionListener" class the JPanel had to be set to a listening state with "setFoucsable" and by adding the listener object to it. The player input sets a value for the speed and once released the speed is set to zero moving the player only when a button is pressed. When the jump button is pressed the vertical speed is calculated and applied but to make sure that the player cannot jump forever a cap was applied to the maximum height that can be reached from the position that the jump was initiated from. The player could not jump again once the maximum height had been reached until contact with ground had been made.

4.2.4 - Creating obstacles and their collisions

To present a challenge for the player obstacles had to be created, there were two different types of obstacles the stationary and projectile. The obstacles were both created with the use of an "arrayList" object which allows items to be flexibly added and removed from a list, as obstacles would be disappearing once they had left the screen and added at random. The draw and move methods for both objects operate in the same manner and are very similar to that of the platforms. The update method is used to move each obstacle but also detect whether the obstacles are out of the games boundary and if they are they will be removed from the list to ensure that the Java garbage collector cleans up the memory that was being used by the obstacles. The stationary objects were created first, they move with the speed of the platforms to make them appear to be stationary while the player ran past. The random chance to create the obstacles was at 1 in every 120 cycles of the game, this meant that roughly every 2 seconds an obstacle would spawn and be added to the game. To make the game more varied the Stationary obstacles have been set to change their height at random every time one is created they can range from 30 pixels in height to 90 pixels in height. The moving object has been implemented slightly differently where it only appears once the player has travelled a value of 500 in distance. The random chance for the moving object to be created is 300 to 1 as it is designed to be more challenging and thus less occurrence was needed. Testing was completed regarding probability for creating the obstacles where both chances to create new obstacles were set to 1 in 120 but it was found that the game was nearly impossible to play and therefore the moving obstacle spawn rate was reduced. The moving object will also change the Y-axis that it is created on due to a random range, between the platforms to the midpoint of the players jump height. Collision detection for the obstacles was different compared to the platforms where for the stationary obstacle the game had to be able to tell if the player was touching it with 3 different sides, while the moving object has be able to tell when the player is touching it on all 4 of its sides and calculate for all of them independently and if so make sure that the end game screen would appear.

4.2.5 - Setting up game ending conditions and Scoring

The final part of the game development involved displaying the final distance that the player achieved as well as a message to let the player know the game was over. This stage began with displaying and calculating the distance, which was set up simply by counting each platform that reached -50 on the X-axis and then adding together the width of those platforms. The distance was then displayed as an image in the top right of the screen by drawing a rectangle to be a box container and simple drawing a string that contained the distance to the screen within the box. The end game scenario was implemented last within the code where once the game had ended it made sure that all game updates as well as rendering would stop. The game ending value would return true when the player character went out with the borders of the game completely or if the player touched an obstacle. The game ending would then trigger the end game message to let the player know that they had lost, and this would be the final part to the game before closing the window and starting again.

4.3 - Game Maker development

The game maker development process was much shorter than that of Java and this is because the creation of all the functionality and methods is already done. The only part that Game Maker leaves for the developer is the logic that the game will follow and how the components will come together. The implementation of the game in Game Maker had to be approached slightly differently due to the different nature of the development methods. Game maker has a set of events which are predefined and whether the developer is using the drag and drop system or the GML language will have to use these events to create a game. The events are essentially reactions to certain situations for example the Create event will run when the object is first created, and the Step event occurs every time the game loop occurs, there are also built in collision events for meeting other objects and events such as alarm events for setting up timers. The use of variables was considerably different in Game Maker where there are only three states a variable can be in, either a "global.variable" which is a variable useable by every instance of an object in the game,

there are variables with no definition these variables act like private variables that can be accessed only by a single instance of an object for example to set gravity in the player object only it would appear like so: gravity = 9.8;. The last type of variable is the temporary variable that uses the "var" keyword to set any following characters as containers of information for example "var number = 42;", to further demonstrate the usage of these variable figure 8 displays each in usage.

```
11 var localVariable = "This cannot work outwith this code file";
12 objectVariable = "This works as a variable only this object can access";
13 global.globalVariable = "This variable can be accessed by any object in the game";
```

Figure 8: Displays the working of the different variable types in Game Maker.

4.3.1 - Creating the world and its Assets

The first stage in Game Maker was to prepare foundation that the game would be built upon, this foundation was achieved by creating a new "Room". The "Room" is essentially the screen that the player will see when they run the game and it is where all new objects that will build up the game will be placed. The Room is a simple object to create, all that is required is that the developer adds a new room to the room tab, then a sprite can be created to become the background image of the Room with the width and height adhering to the guidelines set in the game design. The need for a game loop to be made is non-existent in Game Maker as it has a built-in loop that runs automatically, although what was changed was the FPS which was set to 60 from the default 30. The next stage was to start creating sprites for each component that would be needed in the game, the player, the platform, the stationary obstacle and the moving obstacle. Game maker allows the user to create sprites without any need for the object to created first. The same dimensions were used for each sprite along with similar colours as were in the Java development.

4.3.2 - Object creation and platform implementation

Objects in Game Maker are a much easier to create than in Java where to create the objects required for the Game Maker game simply selecting add new object must be selected. The platform and player object were both created first and they were then dragged and dropped within the Room to add them to the game. Every object that is added to the room was then given two essential events for them to perform actions the create event and the step event. The platform was then given variables in the create event to keep the colour scheme the

same as the java game, where random colours were applied to each platform that was created, this caused many issues as Game Maker does not create random values unless specified to do so with a special method that must be implemented before any random event is called. The platform was then repeatedly added to the room until 17 were present, dragging and dropping the platforms was an easier approach than following the same method that Java used. The step event was then filled with the controls to move the platforms across the screen. Game Maker uses built in constants to control objects and by adding to the X and Y constants every time the step event activates the platform can be moved across the screen.

4.3.3 - Player object and obstacles

To implement the player object, it was dragged from the menu holding all the created objects and placed into the room to ensure that it existed in the game. Then the sprite for the player was then selected from the sprite drop-down list to give the player a form. In the "create event" variables for the player were established for the control of the players actions such as the vertical and horizontal speed as well as the maximum height that can be jumped. In the "step event" to make the controls more alike the java game they were implemented in the same way, where instead of using the built-in events for input a series of "if" statements were used. The way in which the Game Maker game detects if the player is in the air is also similar to the logic followed in the Java development where a Boolean value is returned to check if the player is in the air and if they are as long as the player does not exceed the maximum jump height will remain in the air. The creation of the obstacles was then established through the use of another control object, the "ControlAndScoreObject", that would randomly generate the obstacles as the game continued, this object also kept track of the distance travelled and printed the distance to the screen. The control object, was implemented so that it would generate new instances of the obstacles with the same probability as the Java development. The Draw event was then used in the control object to draw a blue box and then to draw the string containing the distance that had been travelled. The stationary obstacle was then created to be varied in size with the same random range as in the Java development, while the moving Obstacle would have the same random range for potential Y positions as the Java project. The collisions of the obstacles with the player as well as going out with the games boundaries

were all handled within the Player object. The collision events in the player object editor were set to respond if the player collided with the moving or stationary obstacle in which the only command was to set the end game variable to true, which would stop the game from operating and show the end game message. The “outside room” event was also used to test for the player leaving the screen in which the end game variable would also return true.

Chapter 5 – Testing and results

5.1 - Testing methods and approach

5.1.1 - Study design

The primary research will be conducted through a comparative analysis between Game maker and Java. As the comparison of different programming languages using game development has not been researched extensively, there is a gap in the current knowledge that this research intends to fill. A comparative study was selected as it allows for the expansion upon initial knowledge and acts as a foot hold for developing research in each area. The comparative method takes two or more similar cases which have an essential difference and analyses the outcome caused by this difference (Routio, 2007). The results of this study will hopefully determine which of the two game development methods is more suited to the specific area of study.

To make a fair comparison between Java and Game Maker, a game has been developed using each method and thereafter assessed. The games developed will be tested against each other upon completion to assess the relative success of each development method.

The following criteria will be measured for the evaluation and comparison of the games:

- CPU usage – This will measure how much work the computer is doing to calculate the information necessary to run the game.
- GPU usage – If the computer does have a GPU it will assist in the image display and buffering of the game. As with the CPU, the percentage of work done by the computer will be recorded (Mishra and Shrawankar, 2016).
- Memory usage – This equates to the Random-Access Memory (RAM) and will be monitored to measure how much memory the game requires to function.
- File size – This refers to the size of the game in terms of raw data or how much stored memory the game occupies on the computer's hard drive (Stein and Geyer-Schulz, 2013).

The CPU, GPU and Memory usage will be measured with the use of the freeware software HWinfo. HWinfo was selected as it provides "comprehensive hardware analysis" in real time and provides all information necessary for the comparison of each game (HWiNFO, 2017). To make a better comparison when using Game Maker, as much GML code as possible will be used to provide the highest level of control possible and achieve a similar development process to Java. Comparison between the games will also be partially based on the experiences of the developer. To guide the developer's commentary of the development process several factors have been selected for consideration.

The factors that will be considered for the development are as follows:

- Documentation and support – This includes the ease of access to beneficial information (either official or otherwise) regarding each method.
- Development process – Each will be assessed in terms of its ease of understanding, capabilities and weaknesses and strengths.
- Lines of Code – This will help act as a measure of the amount of time and effort required during the creation of the game (Jonsdottir, 2010).

5.1.2 - Procedure

To ensure fairer testing the games a specific approach was created to make completing the testing of the games repeatable in the future.
The testing was completed on a computer with the following specifications:
CPU: Intel i5-6600K
GPU: Nvidia GeForce GTX 950 2G
RAM: DDR4-3200 Corsair 8 GB
Motherboard: ASUS Z170 PRO GAMING

The testing process will be as follows:

1. All components for testing need to be installed on the test computer, the latest version of Java (1.8 or higher), the measurement software HWinfo, the Game Maker game, ensure the Java Game ".Jar" file is available and that a stop watch is ready to monitor time past on each test.
2. Stop any unnecessary processes that the computer is running.

3. Run the HWinfo software then proceed to select the sensors button that should open all processes that are being monitored currently.
4. Within the sensor page proceed to deselect all measurements being taken except for:
    - Physical memory used
    - Physical memory load
    - Total CPU usage
    - GPU D3D usage
    - GPU memory usage
5. Set the interval time for logging data to 0.5 seconds by opening the settings menu depicted as a "cog".
6. Select "start logging" on the sensor sheet and 10 seconds later begin playing the Game Maker game, once the game has been played and lost stop logging the information, make sure at least 20 seconds of gameplay is recorded.
7. Log the excel file under the development method name and then the test number, for example 'GameMakerTest01'
8. The amount of time taken to start and finish the Game Maker game should be replicated for the playthrough of the Java game by either purposefully losing the game or terminating the program.
9. Log the excel file under the development method name and then the test number, for example 'JavaTest01'
10. Repeat steps 6 to 9 three times to show whether data anomalies will appear in the data received.
11. Move all data into a new excel sheet to begin analysis.

5.1.3 - Analysis

The raw data from the initial tests will be presented in an Excel work sheet and for each test cycle the Java and Game Maker game data will be transferred into the same file to prepare for analysis. The three files containing data for each cycle of testing were analysed through Excel by producing line graphs to plot the changes in data over time. The average values for

each set of data was produced using Excels built in formula for generating averages, these values were then used to create a clearer depiction of which method was more effective.

5.2 - Results

The main components of each Game Maker and Java game have each been assessed, and the results have been displayed in a line graph fashion. Assessment was completed on four performance-based factors and one development-based criteria. The development testing was completed three times to produce a greater pool of result and thus fairer analysis where total CPU usage, RAM usage, GPU physical memory usage and GPU D3D usage was all measured.

5.2.1 - Total Computer Processing Unit Usage

The figures labelled from 9 to 11 all depict the measure of the total CPU usage that each version of the game uses. The graphs could indicate that the usage is even where the Game Maker version uses larger amounts of the CPU at times then activity dips and the Java version stays near the same level with smaller peaks and troughs of CPU usage. The average values, which can be seen in figure 12, obtained from each test cycle all determine that the Game Maker version of the game uses less CPU usage than the Java version.

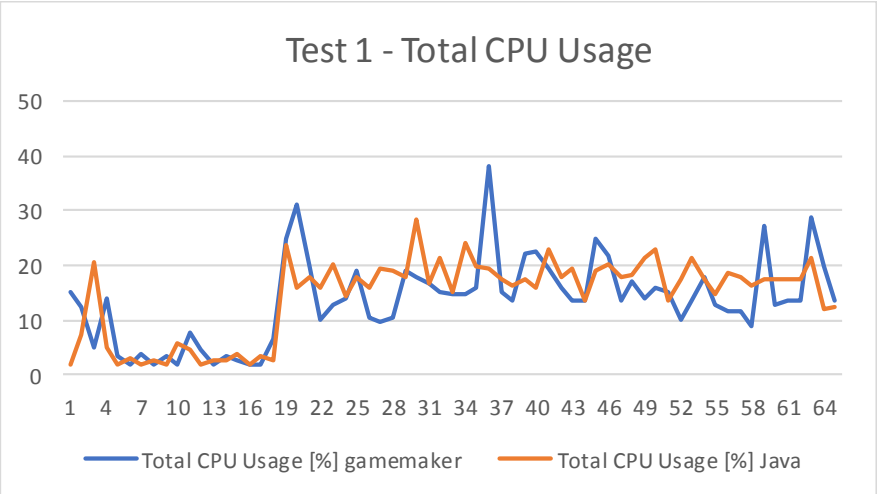


Figure 9: The first test completed for receiving the CPU usage while playing both Game Maker and Java

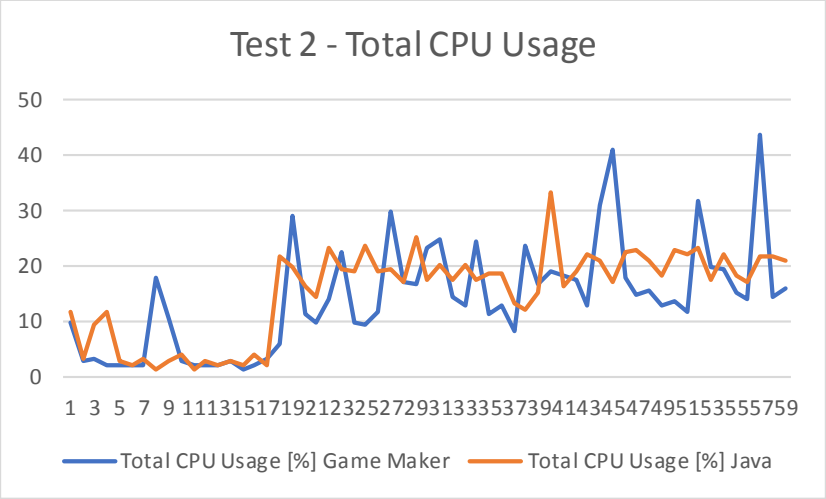


Figure 10: The second test completed for receiving the CPU usage while playing both Game Maker and Java.

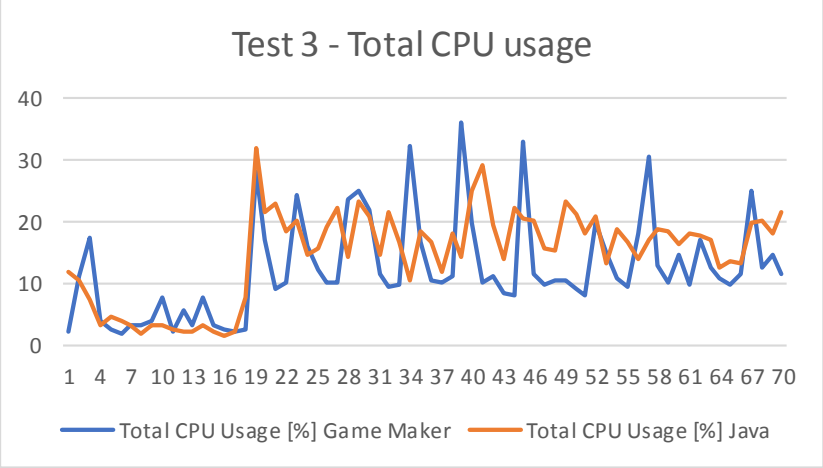


Figure 11: The third test completed for receiving the CPU usage while playing both Game Maker and Java.

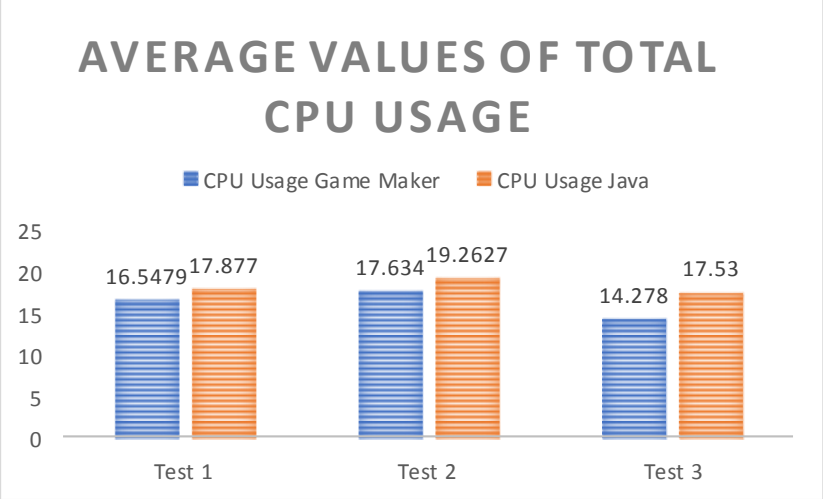


Figure 12: The average values is depicted for each development method for all three cycles

5.2.2 - RAM Testing

The physical memory usage (aka RAM) tests can be seen in Figure 13 to 15 where the comparison in Kilobytes over half second increments is made for each Game maker and the Java game version. The first test cycle displays a large difference between the two different methods that then also remain at a steady rate of memory usage, while the other two tests completed displayed different results that suggest the memory usage is much more similar. The second and third test also show that the memory usage of the Java and Game Maker game increments together as play continues. The average values in figure 16 depict that with the discrepancy of the first test the physical memory used is almost undiscernible.

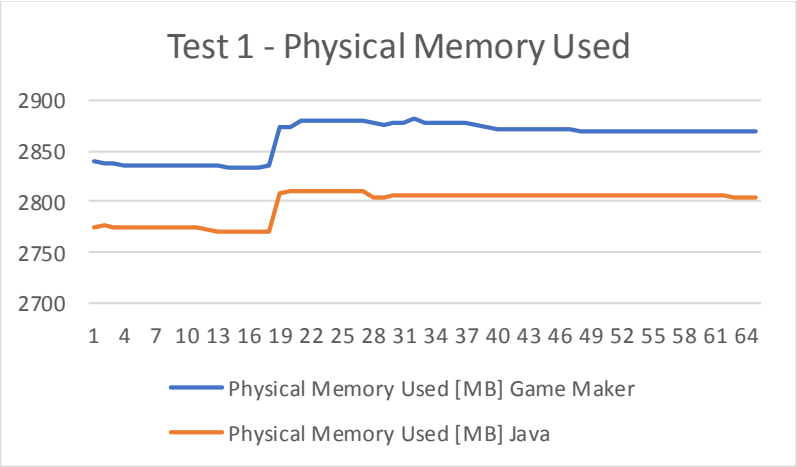


Figure 13: The first test of the RAM usage between Game maker and Java.

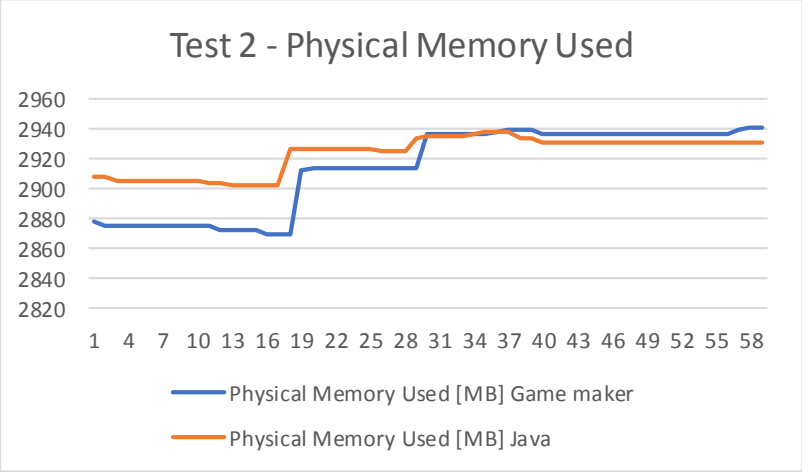


Figure 14: The second test of the RAM usage between Game maker and Java.

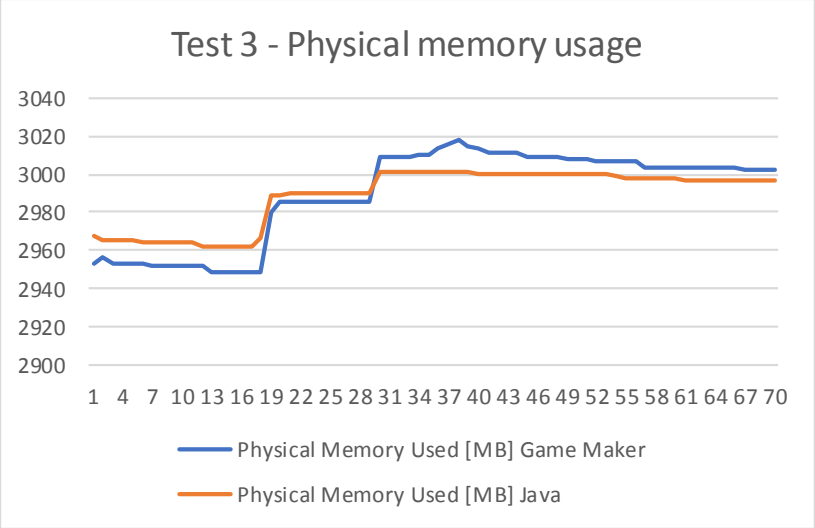


Figure 15: The third test of the RAM usage between Game maker and Java.

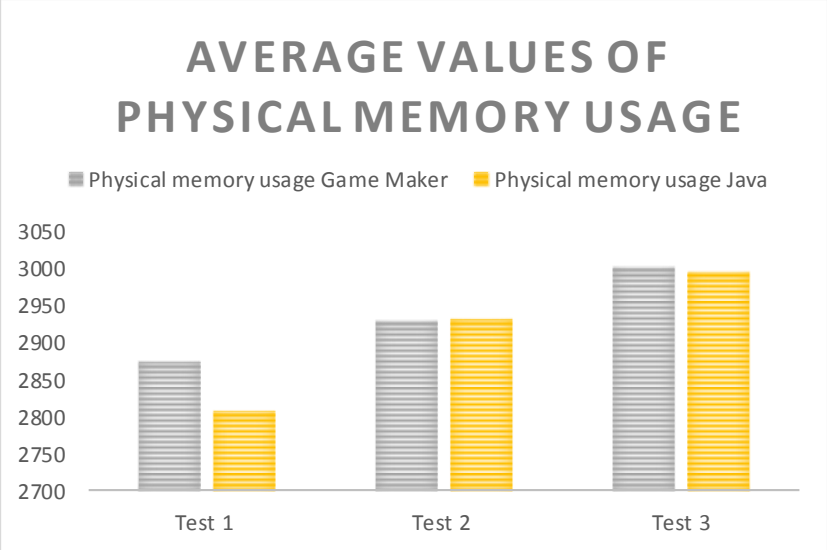


Figure 16: The average value of physical memory usage over time, comparing Java and Game Maker

5.2.3 - Graphical Processing Unit Memory Usage

The GPU's memory usage of each game development is displayed from figure 17 to 19 where the percentage of usage for each game is displayed over time. The GPU memory usage figures each depict that the Game Maker game required more memory than that of the Java game. The values received in each test all depict a spike and then a leveling off and plateauing of the data received. The average value received from each of the test cycles, depicted in figure 20, adds no new data from what was already known where each test is showing a similar result.

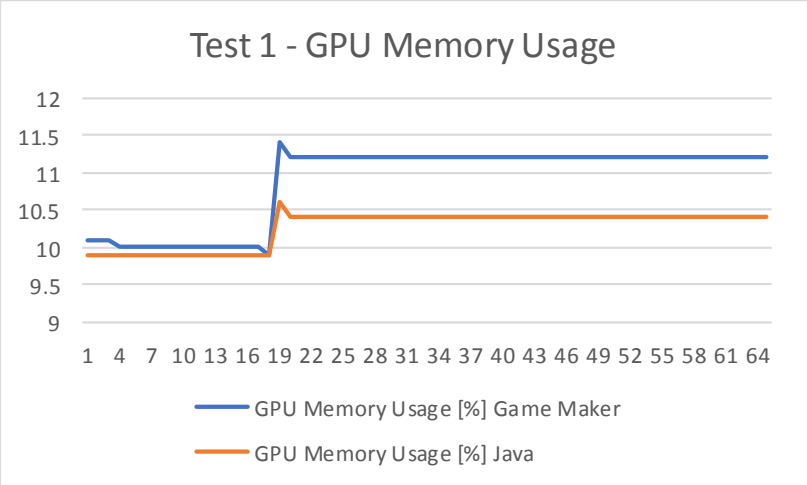


Figure 17: The first test regarding the Graphics processing unit's memory usage.

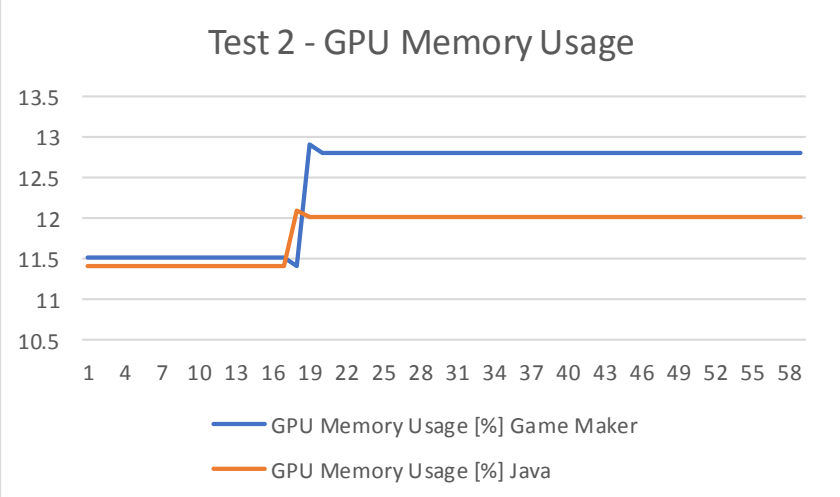


Figure 18:The second test regarding the Graphics processing unit's memory usage.

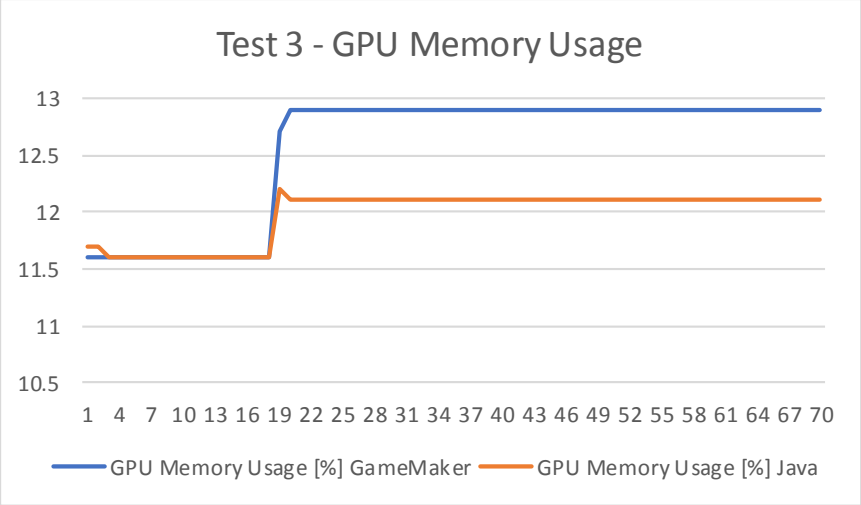


Figure 19: The third test regarding the Graphics processing unit's memory usage.

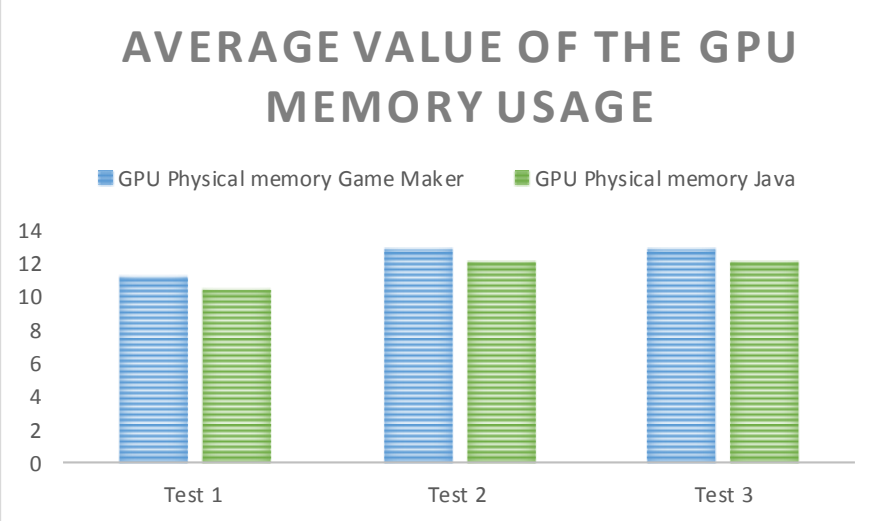


Figure 20: The average value of the GPU's memory usage over time that each development method required

5.2.4 - Graphical Processing Unit D3D Usage

The GPU D3D usage is shown and tested within the figures 21 to 23 where the usage is tested across time. The D3D usage is the amount of effort the GPU uses to process data using DirectX or Direct3D which then allow the image being processed to be rendered. The tests depict the usage as an erratic process that has sharp spikes of D3D usage then plateauing at a lower value. The average values for D3D usage in figure 24 demonstrate that overall Game Maker uses less D3D usage as shown in the first and third tests average results, although Java uses less D3D usage in the second test but only marginally.

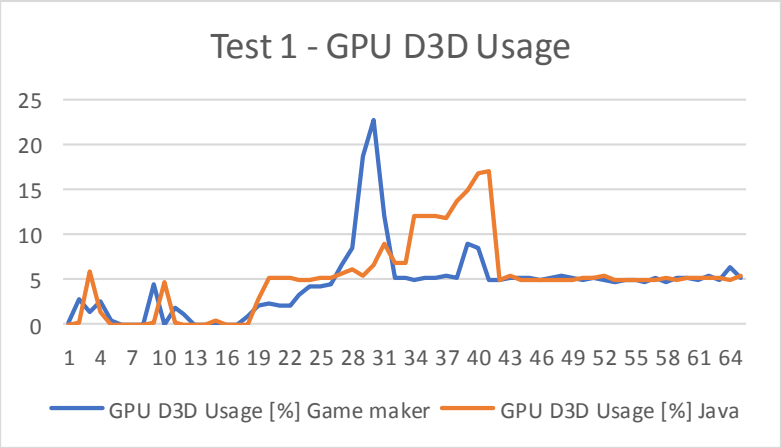


Figure 21: The first test regarding GPU D3D usage.

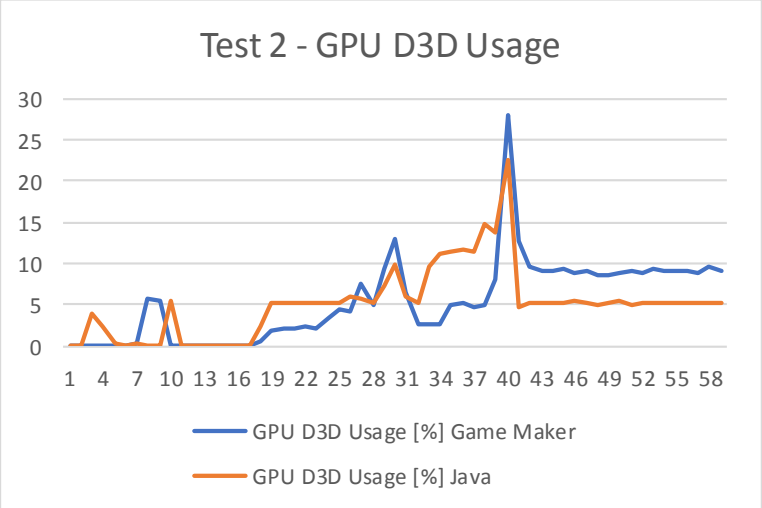


Figure 22:The second test regarding GPU D3D usage.

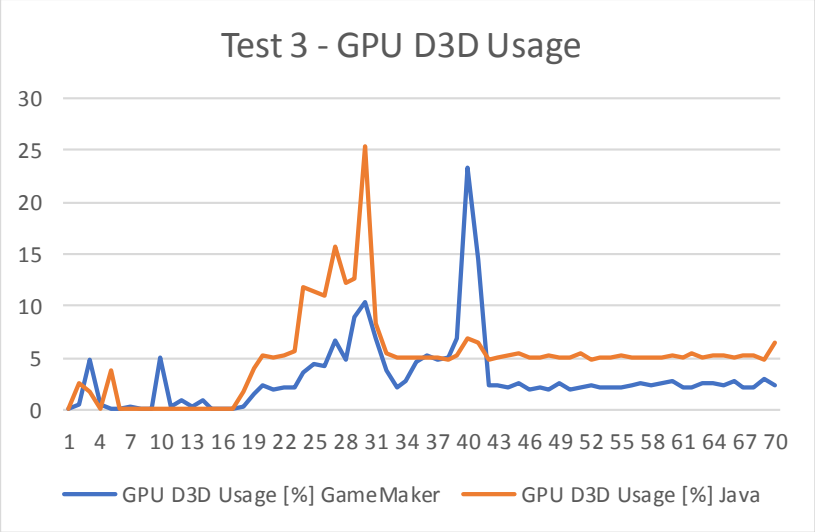


Figure 23: The third test regarding GPU D3D usage.

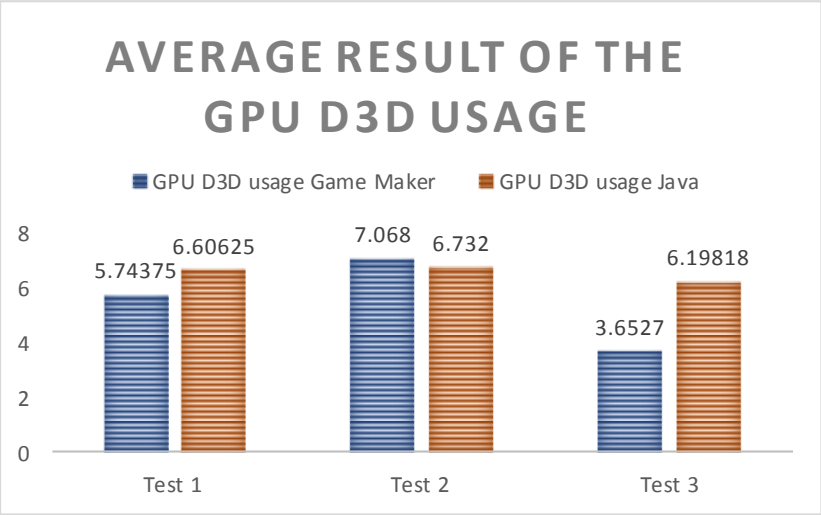


Figure 24: The average value obtained from the GPU as it processes and renders the game with DirectX or Direct3D

5.2.5 - Measuring File sizes between the Game maker and the Java games

The final performance-based result is that of the difference in files size (shown in figure 25) between the two completed games and their supporting folder that they require to run. The Game Maker version is clearly much larger that the Java version of the game. The Java game comes to a total of 76 Kilobytes while the Game Maker game is 4.36 Megabytes which is over 54 times bigger.

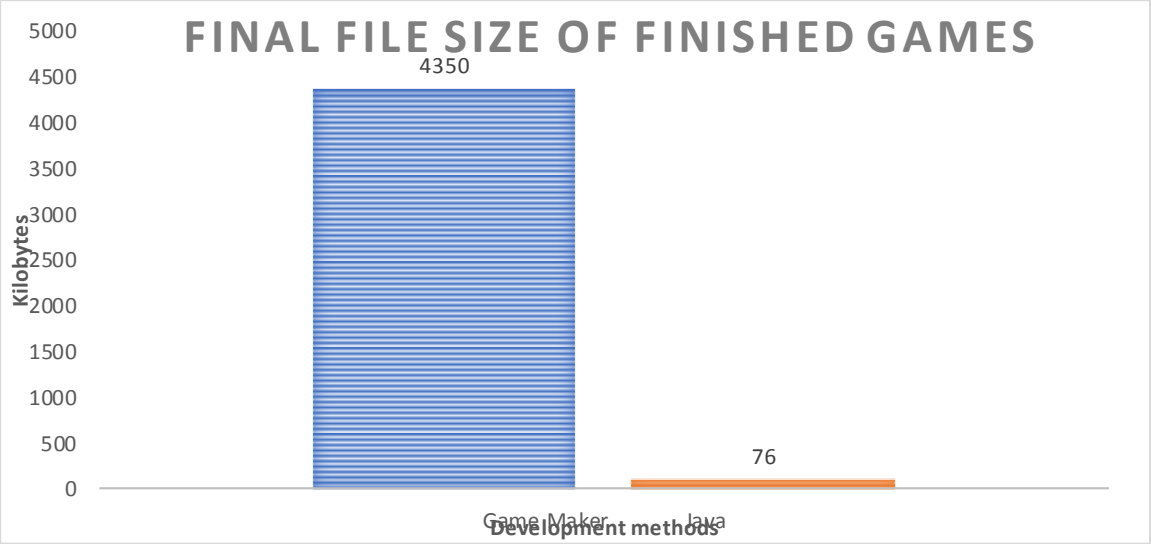


Figure 25: The file size of the finished game and supporting folders that allow the game to operate, measured in Kilobytes

5.2.6 - Lines of code Measured

In figure 26 the amount of lines is measured that were used to create each game. The creation of the Java game required the use of 661 lines of code while the Game Maker development only required 118 to complete the same game.

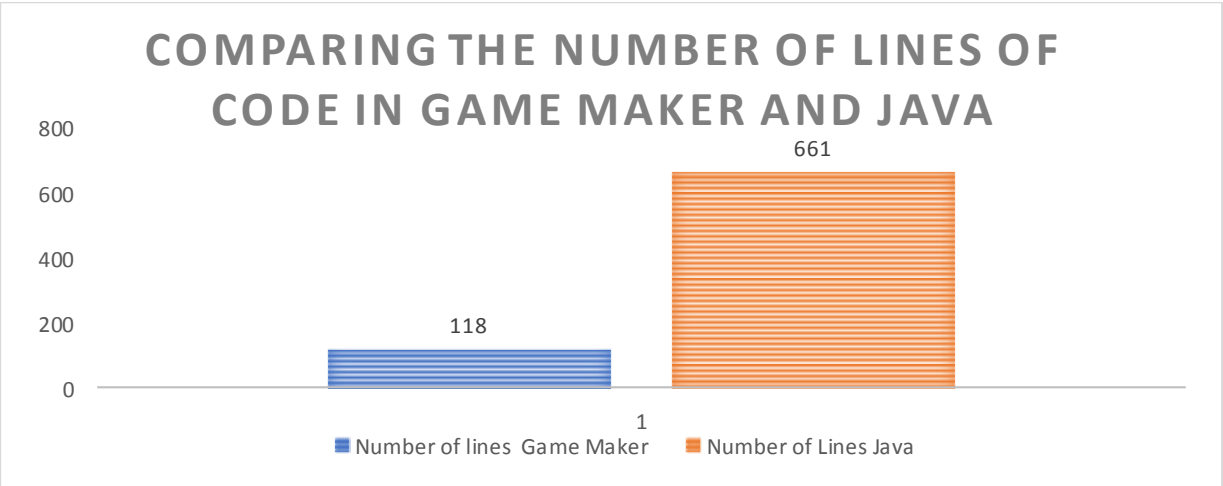


Figure 26: The number of lines of code that has an operation, not including blank lines or comments.

Chapter 6 – Summery

6.1 – Conclusion

The results presented from figure 9 to figure 26 all demonstrate the performance of Java against Game Maker and each figure displays the game being played over time. The results clearly depict the point at which the game is initiated by the spike of activity in each graph which made analysis much easier when comparing the results and creating average values. The CPU usage for both versions of the game show a very similar line of usage. In each test completed Java seems to hold a more uniform peak and trough of processing power usage while Game Maker fluctuates more violently but still seems to produce a similar amount of CPU usage to Java. The tests regarding average CPU usage in figure 12 display more clearly the results and determine that regarding CPU usage Game maker is more efficient, with each test cycle concluding the same result. The expected result for the CPU usage was that Java would be the more efficient platform, as it has less abstraction from the hardware but as shown in the results Game maker has proved to be more efficient. The potential reasoning for Game Makers greater efficiency with CPU usage could stem from the fact that Game Maker was created by team of professionals while the Java game was created by a Novice. The tests for RAM usage were found to be very similar and the averages demonstrate this similarity in figure 16 where test 2 and 3 are very similar results where neither Game Maker or Java is could be considered superior. The physical memory's, RAMs, usage is an important factor for game designers as it will limit the players access to a game as a large demand on the RAM can make games operate at much slower rates but with no definite result neither Game Maker or Java can be shown to be superior. The GPU usage results were split into two sections, the amount of memory used and the GPU D3D usage. The GPU memory usage displayed in figure 17 to 20 was uneventful with similar results for each test, Java is shown to clearly require less physical memory on the GPU while Game maker is seen to be less efficient and require more. The GPU memory usage is an important factor to consider where it can reference how much data needs to be stored while the game is being played and as the RAM (Physical Memory) did not return any significant results the

GPU memory can be regarded as more important. The GPU D3D was used to measure how much work was being done by the GPU to process the games being played. The result of the GPU D3D average, in figure 24, displayed that most of results implied that Java needed more processing power in order to operate the game and therefore Game Maker was superior for the GPU D3D processing. The final factor that was considered for the performance-based results was that of the final file size of each game. The file size is shown in figure 25 where the Java game is 76 KB and the Game Maker game is 4.35MB, this difference in size Is very significant as the game that has been created in this study is very simplistic. The file size could become a problem if a game were to be created on a larger scale, using Java and Game Maker, with more advanced content and programming techniques and the file size also scaled where a 500MB Java Game could potentially be a 28.5 GB Game Maker game, although this type of scaling would likely not be the case. The huge difference in file size would be okay for smaller games but for larger games developers would not wont their players to be inconvenienced by a large game. The performance-based criteria were do not have a definite answer where Game maker was superior for processing efficiency while Java performed better with regards to memory usage and file size. The argument can be made however that the processing power required to run the game is more important, as memory can be augmented while users will try to keep the CPU and GPU for as long as possible before upgrading and in this case Game Maker proves itself to be a better game development method.

The documentation available for each method varied greatly where Java has a large community base with stack overflow as one of the largest forums for posting and answering programming related questions and Game Maker has the Game Maker Community which is a forum hosted by YoYo games. The source files for Java are all available for access to understand any method or object that is built into the Java. Game Makers documentation mostly consist of the online manual that is supplied by YoYo games and if a GML function is searched using for example google, the online Game Maker manual will appear, it does provide examples of use, but it is not always clear on how to use some of the functions and it is outdated for some methods (Docs2.yoyogames.com, 2018). The Documentation of Java may take longer to understand due to the complex nature of some of the objects and methods whereas Game Maker simply provides the method that is being searched and

examples of use to make implementation and understanding easier for the developer. The support that is available for Game Maker with regards to the developing community is poor unlike Java whose users are vast and very active and almost any question has already been asked and answered in a professional manner. The development process for Game Maker and Java varied greatly and will likely be one of the most influential factors for new developers. The game making process in Java required prior knowledge of Java, an understanding of Object-Orientated programming and understand how game development is approached specifically in Java whereas, Game Maker can be used by first time novices with no previous programming knowledge especially as the drag and drop system can be used instead of writing GML code. The amount of time and effort put into developing each game has a considerable gap and can be demonstrated with how many significant lines of code were used to create each game. The lines of code in the game development vary greatly, as seen in figure 26, where 661 lines of code were used to create the Java game while only 118 lines of code were used in the Game Maker development. The huge difference in lines of code created only gives an indication as to the time put into each game. In Java there is no visual aspect of the games creation the developer must have a full understanding of what they are creating and exactly how it needs to work, while in Game Maker each object has a visual representation which can be move around the Room editor at will. The control that Game Make provides over positioning and the visualization of objects makes understanding the creation of the game much simpler.

The analysis of each factor considered has made it clear that when it comes to development Java and Game Maker both have their advantages where Game Maker is more efficient for processing power while Java creates smaller files sizes and is more efficient for GPU memory usage. The better of the two methods in terms of performance cannot be determined effectively. The development of each method however showed that although Game maker does not have documentation as comprehensive as Java and has smaller community support it does have considerably faster development times as well as very simple and intuitive user interface designed for beginners. The final result of this study concludes that Game Maker is the superior method of development for 2D games for those new to game development.

6.2 - Critical Self Appraisal

6.2.1 - Project review

The project presented many challenges throughout, where research was producing little results as the topic is not widely researched or the development process was not moving ahead as scheduled due to errors. The projects conclusion was correct as it could be according to the results found in the testing and measures taking in the development. Meeting critical points in the development was found to be challenging as few sources for code as possible were used, this was done as the development aimed to be as original as possible. In the Java development for example, the only code which required aid was that related to receiving player input as well as the implementation and structure of the game loop. The development duration for each game was significantly different. Specifically, implementing the Java game and understanding the logic that was required to be applied took several weeks. However, once the games logic had been understood, the majority of the Game Maker development was created in 2 days. Results were found to be difficult to obtain as the games were required to be played for almost the exact same duration in order to compare them. A reoccurring issue experienced during testing was that the game would be lost too quickly meaning that recording had to be stopped and all results that had been saved deleted. Tests then had to be restarted until the correct amount of time had been recorded while playing the game. The analysis of each set of data was limited, where tools for more advanced analysis were not available. As well as tools not being available, obtaining a more in-depth analysis of results, more data would be required and for this to be possible the game would have to be tested many more times and for a longer duration which could not be achieved in the available time frame.

6.2.2 - Limitations

There were many limiting factors that prevented the project from reaching its full potential. These factors occurred throughout the study in the development, testing and the analysis stages. The development of the present study is the most important component and has the greatest impact on the conclusion.

The most significant factor affecting the development of the games was the developer's knowledge, where the development relied heavily on the programming abilities of the developer. The developer is still a relative novice in Java programming especially in game development, as Java requires more in-depth knowledge than Game Maker which is specifically designed for beginners. To reduce the error in results potentially caused by poor programming, a professional developer could be employed to create a more efficient game in both Java and Game Maker, therefore making results regarding the final development more trustworthy. The development also produced issues with regards to the Game Maker development where Game Maker studio 2, which was used for the development process, could not create an executable file as the trial version was used. To create the final executable for the Game Maker development an older version of Game Maker was used (Game Maker studio 1.4) and the Game Maker studio 2 version was converted back to 1.4, where several lines of code were adopted to allow for the old version to compile. The effect that converting the Game Maker version had on the final game and the testing is unclear. The time available to create the project also presented its own issues in development where features such as images and animation could not be developed in time. The addition of the methods to incorporate animation in Java was not possible with the time remaining.

The testing process also presented several limiting factors which effected results achieved. One of the main limitations of the development testing was methods that were used for measurement. For example, when starting and stopping the logging of data, a stop watch was used and once the recording was terminated, the data was switched off by physically stopping the logging of data. Recording data in this way means that it can be impacted by human error and thus extra data gained had to be edited to match with the counterpart test. This loss of data may affect the results through potentially altering average values that were obtained. Each game was tested for only 20 seconds as the game became too fast to play for any longer than this duration. Given the small sample size for each test, these therefore had to be very accurate and many different factors affected this accuracy. The games obstacle creation worked on a probability basis where potentially no obstacles would be created but during other play attempts, too many would appear. The difference in spawning obstacles could potentially change the amount of usages that were reported in each test and therefore change the results and outcomes. The potential for other computer processes starting while one of the games was being played is an important factor to be

considered. In Figure 13 the physical memory shows a clear distinction between the two methods yet in the other test completed, this distinction in usage was not observed. Thus, it can then be concluded that an error occurred which affected the memory usage for the Figure 13 test. The limiting factors for testing could be addressed by a number of mechanisms from altering the game to make the play testing last longer to and gain more results or by increasing the number of tests completed. The use of software that begins monitoring for specific programs, allowing the games to be tested independently of other computer processes would greatly improve the quality of the results achieved from the testing.

6.2.3 - Future work

There is great potential for more research to be completed in the same area of study using the same comparative methods in order to create a more comprehensive knowledge base. Future research should compare multiple game engines and programming languages from the top of their respective fields for a more in-depth study.

References

Batchelor, J. (2013). *Hotline Miami bags most nominations in 2013 Golden Joystick Awards*. [online] MCV UK. Available at: http://www.mcvuk.com/news/read/hotline-miami-bags-most-nominations-in-2013-golden-joystick-awards/0120549 [Accessed 10 Mar. 2017].

Blog.rogach.org. (2015). *How to create your own simple 3D render engine in pure Java*. [online] Available at: http://blog.rogach.org/2015/08/how-to-create-your-own-simple-3d-render.html [Accessed 15 Mar. 2018].

Capello, D. (2017). *Aseprite*. [online] Aseprite.org. Available at: https://www.aseprite.org/ [Accessed 13 Oct. 2017].

Christopoulou, E. and Xinogalos, S. (2017). Overview and Comparative Analysis of Game Engines for Desktop and Mobile Devices. *International Journal of Serious Games*, [online] 4(4). Available at: http://journal.seriousgamessociety.org/index.php?journal=IJSG&page=article&op=view&path%5B%5D=194&path%5B%5D=pdf_104 [Accessed 17 Mar. 2018].

Cohn, M. (2017). *Scrum Methodology and Project Management*. [online] Mountain Goat Software. Available at: https://www.mountaingoatsoftware.com/agile/scrum [Accessed 19 Nov. 2017].Comer, D. (2017). *Essentials of Computer Architecture*. 2nd ed. CRC press, pp.163-165.

Davison, A. (2005). *Killer game programming in Java*. 1st ed. O'reilly.

Docs2.yoyogames.com. (2018). *GameMaker Studio 2*. [online] Available at: http://docs2.yoyogames.com/ [Accessed 19 Mar. 2018].

Doman, M., Sleigh, M. and Garrison, C. (2015). Effect of GameMaker on Student Attitudes and Perceptions of Instructors. *International Journal of Modern Education and Computer Science*, [online] 7(9), pp.1-13. Available at: http://www.mecs-press.org/ijmecs/ijmecs-v7-n9/IJMECS-V7-N9-1.pdf [Accessed 8 Mar. 2017].

Fingas, J. (2017). *Indie game award finalists include 'Hyper Light Drifter' and 'Inside'*. [online] Engadget. Available at: https://www.engadget.com/2017/01/09/independent-games-festival-award-nominees-2017/ [Accessed 11 Mar. 2017].

Foundation, K. (2017). *Krita | Digital Painting. Creative Freedom.*. [online] Krita.org. Available at: https://krita.org/en/ [Accessed 15 Oct. 2017].

Guins, R. and Lowood, H. (2016). *Debugging game : a critical lexicon*. 1st ed. p.203.

Habgood, J. and Overmars, M. (2013). *The game maker's apprentice*. 1st ed. [Berkeley, Calif.]: Apress.

Hoganson, K. (2010). *Teaching programming concepts with GameMaker*. [online] Dl.acm.org. Available at: http://dl.acm.org/citation.cfm?id=1858610 [Accessed 27 Feb. 2017].

HWiNFO. (2017). *HWiNFO - Hardware Information, Analysis and Monitoring Tools*. [online] Available at: https://www.hwinfo.com/ [Accessed 11 Mar. 2017].

Irimia, A. (2001). *ENHANCING THE INTRODUCTORY COMPUTER SCIENCE CURRICULUM: C++ OR JAVA?*. 1st ed. [ebook] Available at: https://pdfs.semanticscholar.org/2887/32baac065bc8d39a0cb23b2b0b3174529770.pdf [Accessed 5 Mar. 2017].

Jonsdottir, R. (2010). *A comparison of game engines and languages*. 1st ed. [ebook] Available at: http://skemman.is/stream/get/1946/5782/14861/1/Final_Full.pdf [Accessed 8 Mar. 2017].

Kalderon, E. (2011). *Game engines: What they are and how they work*. [online] Eyal Kalderon. Available at: https://nullpwd.wordpress.com/2011/05/09/game-engines-what-they-are-and-how-they-work/ [Accessed 28 Feb. 2017].

Koch, A. (2011). *12 Advantages of Agile Software Development*. 1st ed. [ebook] Available at: https://cs.anu.edu.au/courses/comp3120/public_docs/WP_PM_AdvantagesofAgile.pdf [Accessed 15 Mar. 2017].

Koenig, M. (2017). *Free Sound Clips | SoundBible.com*. [online] Soundbible.com. Available at: http://soundbible.com/ [Accessed 12 Mar. 2017].

Lange, A. (1996). *Statistical yearbook, 1994-1995*. 1st ed. Strasbourg, France: European Audiovisual Observatory, p.123.

Lexaloffle.com. (2017). *PICO-8: FANTASY CONSOLE*. [online] Available at: https://www.lexaloffle.com/pico-8.php [Accessed 21 Nov. 2017].

Lowood, H. (2014). *Game Engines and Game History*. 1st ed. [ebook] p.181. Available at: http://www.kinephanos.ca/2014/game-engines-and-game-history/ [Accessed 1 Mar. 2017].

Madhav, S. (2013). *Game Programming Algorithms and Techniques: A Platform-Agnostic Approach*. 1st ed. Addison-Wesley, pp.2-15.

Mangione, C. (1998). *Performance tests show Java as fast as C++*. [online] JavaWorld. Available at: http://www.javaworld.com/article/2076593/performance-tests-show-java-as-fast-as-c--.html?page=2 [Accessed 7 Mar. 2017].

Mannila, L. and de Raadt, M. (2006). *An objective comparison of languages for teaching introductory programming*. [online] Available at: http://dl.acm.org/citation.cfm?id=1315811&CFID=910804528&CFTOKEN=92176813 [Accessed 11 Mar. 2017].

Mishra, P. and Shrawankar, U. (2016). Comparison between Famous Game Engines and Eminent Games. *International Journal of Interactive Multimedia and Artificial Intelligence*, 4(1), p.69.

Nandy, A. and Chanda, D. (2016). *Beginning Platino Game Engine*. 1st ed. Berkeley, CA: Apress, pp.1-10.

Parveen, Z. and Fatima, N. (2016). *Performance Comparison of Most Common High Level Programming Languages*. [ebook] Hail: © MEACSE Publications, pp.246-257. Available at: http://www.meacse.org/ijcar/archives/109.pdf [Accessed 15 Mar. 2018].

Pavkov, S., Franković,, I. and Hoić-Božić, N. (2017). *Comparison of Game Engines for Serious Games*. [ebook] Opatija, pp.728-732. Available at: http://ieeexplore.ieee.org/document/7973518/ [Accessed 15 Mar. 2018].

Playitagainproject.org. (2013). *The Life and Times of an 80's Game Programmer – Putting it all together | Play It Again*. [online] Available at: http://playitagainproject.org/the-life-and-times-of-an-80s-game-programmer-putting-it-all-together/ [Accessed 15 Mar. 2017].

Routio, P. (2007). *Comparative Study*. [online] Uiah.fi. Available at: http://www.uiah.fi/projekti/metodi/172.htm [Accessed 10 Mar. 2017].

Salen, K. and Zimmerman, E. (2011). *Rules of play*. 1st ed. Johanneshov: TPB.

Sawer, C. (2005). *Chris Sawyer Software Development*. [online] Chrissawyergames.com. Available at: http://www.chrissawyergames.com/info.htm [Accessed 14 Mar. 2017].

Sheffield, H. (2015). *The man who sold his startup for $2.5b says it's made him miserable*. [online] The Independent. Available at: http://www.independent.co.uk/news/business/news/the-man-who-sold-minecraft-to-microsoft-for-25-billion-says-its-made-him-miserable-10479865.html [Accessed 8 Mar. 2017].

Stein, M. and Geyer-Schulz, A. (2013). A Comparison of Five Programming Languages in a Graph Clustering Scenario. *Journal of Universal Computer*, [online] Vol 19(no 3), pp.428-456. Available at: https://pdfs.semanticscholar.org/8a29/f5468e075f32484ce80025fbef9fdcec934e.pdf [Accessed 10 Mar. 2017].

Techwalla. (n.d.). *Advantages & Disadvantages of High- & Low-Level Language | Techwalla.com*. [online] Available at: https://www.techwalla.com/articles/advantages-disadvantages-of-high-low-level-language [Accessed 15 Mar. 2017].

The Escapist. (2017). *The Escapist : Forums : Gaming Discussion : Poll: Easiest video game genre to make a good game?*. [online] Available at: http://www.escapistmagazine.com/forums/read/9.378843-Poll-Easiest-video-game-genre-to-make-a-good-game [Accessed 28 Feb. 2017].

Tiobe.com. (2017). *TIOBE Index | TIOBE - The Software Quality Company*. [online] Available at: http://www.tiobe.com/tiobe-index/ [Accessed 3 Mar. 2017].

UK VIDEO GAMES FACT SHEET. (2017). 1st ed. pp.2-19.

Ward, J. (2008). *What is a Game Engine?- GameCareerGuide.com*. [online] Gamecareerguide.com. Available at:

http://www.gamecareerguide.com/features/529/what_is_a_game_.php?page=2 [Accessed 30 Feb. 2017].

Yoyo Games. (2017). *GameMaker | YoYo Games*. [online] Available at: http://www.yoyogames.com/gamemaker [Accessed 5 Mar. 2017].

Zechner, M. (2013). *libgdx*. [online] Libgdx.badlogicgames.com. Available at: https://libgdx.badlogicgames.com/documentation.html [Accessed 5 Mar. 2017].

Appendix – A

Java Game development – code by class

mainControl:

```
package control;

import window.scribbleFrame;

/**
 * Created by Matthew on 14/01/2018.
 */
public class mainControl {

    public static void main(String args[]){
    scribbleFrame sf = new scribbleFrame();
    }
}
```

controlJoin:

```
package control;

import objects.Platform;
import objects.Runner;
import objects.movingObstical;
import objects.stationaryObject;
import window.scribbleFrame;

import javax.swing.*;
import java.awt.*;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.util.ArrayList;
import java.util.Random;

/**
 * Created by Matthew on 14/01/2018.
 */
public class controlJoin extends JPanel implements Runnable{
    //game control START//

    private  boolean loop = true;
    private  boolean isPaused = true;
    private  boolean endGame = false;
    private boolean running = false;

    //game control END//
    //Graphics start//

    private Graphics dbg; //double buffered graphics
    private Image dbImage; //image used to put onto the greaphics objects
    private scribbleFrame SFrame;

    private int width, height;
    //Grapgics END//


    //Main Game Loop Variables//
    private Thread gameControl;
    private int sleep;
    private long beforeTime, afterTime,currTime,delay,addExcess,timeDiff,sleepTime;
```

```
    private long overSleepTime = 0;
   private int noDelay = 0;

   private final int NO_DELAYS_PER_YIELD = 10;
   private final int MAX_FRAME_SKIPS = 5;
   private long excess=0L;
   private long period =16666666L;// for 30fps = 0.0333333333, for 60fps =
0.01666666666
    //1.3333333334
    private int framesSkipped = 0;

   //------game objects and items -------//
    private Runner player;
    private Platform pTest [] = new Platform[17];
    private ArrayList<stationaryObject> S_O_List = new
ArrayList<stationaryObject>();
    private ArrayList<movingObstical> M_O_List = new ArrayList<movingObstical>();
    private stationaryObject sO;
    private movingObstical mO;


    private int distance;
    private int score;
    //player speed
    private int speed;
    private int GameSpeed = 1;
    //-------END MGLV-------//


    public controlJoin(int Width, int Height){
        this.height = Height;
        this.width = Width;

        this.setPreferredSize(new Dimension(width,height));
        this.setBackground(Color.blue);
        this.setVisible(true);

        int Xpos = -50;
        int CurrXpos = Xpos;
        int Ypos = 480;
        int PlatWidth = 50;

        player =  new Runner(width/2, height/2,0,0 ,40,70);

        for(int i = 0;i<pTest.length;i++){
            pTest[i] = new Platform(CurrXpos,Ypos,player);
            CurrXpos+=PlatWidth;
        }


        this.setFocusable(true);
        KeyListener keyEvent = new Listener();
        this.addKeyListener(keyEvent);


    }

    public class Listener implements KeyListener{

        @Override
        public void keyTyped(KeyEvent e){

        }

        @Override
        public void keyPressed(KeyEvent e) {
            //set player move speed to acertain value create a limit on it.
```

```
            int keyCode = e.getKeyCode();
            switch(keyCode){
                case KeyEvent.VK_W:
                    player.setVspeed(-15 - GameSpeed );
                    break;
                case KeyEvent.VK_A:
                    player.setHspeed(-9);
                    break;
                case KeyEvent.VK_S:
                    //change to ducking or spinning
                    //could set the position of the player to represent it at a 90
degree angle.
                    break;
                case KeyEvent.VK_D:
                    player.setHspeed(9 + GameSpeed);
                    break;
            }
        }

        @Override
        public void keyReleased(KeyEvent e) {
            int keyCode = e.getKeyCode();
            switch(keyCode){
                case KeyEvent.VK_W:
                    player.setVspeed(0);
                    break;
                case KeyEvent.VK_A:
                    player.setHspeed(0);
                    break;
                case KeyEvent.VK_S:

                    break;
                case KeyEvent.VK_D:
                    player.setHspeed(0);
                    break;
            }
        }
    }


    public void addNotify(){
        super.addNotify();
        startThread();
    }


    public void startThread(){
      if(!running || gameControl ==null) {
          gameControl = new Thread(this);
          gameControl.start();
      }
    }

    public void run(){
            running =  true;
            beforeTime = System.nanoTime();
                while(loop){
                    update();
                    gameRender();
                    activePaint();
                afterTime = System.nanoTime();
                //conversion from nano to milli then to the second time
                timeDiff = (afterTime - beforeTime);
                sleepTime = (period - timeDiff)-overSleepTime;//oversleeptime to
catch up for lost time previously
                if(sleepTime>0) {
```

```
                    try {
                        Thread.sleep(sleepTime / 1000000L);
                    } catch (InterruptedException e) {
                        overSleepTime = (System.nanoTime() - afterTime) -
sleepTime;
                    }
                }
                else{ //sleepTime<=0 it took too long for the program to completer
to comply for the FPS
                    overSleepTime = 0L;
                    excess -= sleepTime;
                    if(++noDelay >= NO_DELAYS_PER_YIELD){
                        Thread.yield();
                        noDelay = 0;
                    }
                }
                beforeTime = System.nanoTime();
                int skips=0;
                while((excess > period)&&(skips < MAX_FRAME_SKIPS)){
                    excess-=period;
                    update();
                    skips++;
                }
                framesSkipped += skips;
        }
    }//end of run



    public void update(){
     if(!endGame) {//is the game still running or not

         if (distance > 500) {
             speed = distance / 500;
            // System.out.println("The new game speed will be : " + speed + "\n the
distance : " + distance);
             GameSpeed = speed;
             setNewGameSpeed();

         }
         CurrentCollide(); // check to see if the player is on the ground /
colliding with the ground
//-----------------------------------------------------------------------------
-------------------------------------------------------------------------------//
         //based on a random chance place a obsticle on a platform, with a maximum
of three
         player.move(GameSpeed);
         for (int i = 0; i < pTest.length; i++) {
             pTest[i].move(GameSpeed);

             //Count for every complete cycle of
             if (pTest[i].getXpos() == 799) {
                 distance += pTest[i].getWIDTH();
             }
         }


         //add obsticles that will either be stationary or moving depending on the
game and how far along it is. First 100 points only ground based objects and then
         //which will also be increased in speed.

         Random r = new Random();


             int i = r.nextInt(120);
             if (i == 1) {
```

```
                    S_O_List.add(sO = new stationaryObject(799, 480, GameSpeed));

            }



        if (distance >= 500) {

            int MO = r.nextInt(300);
            int randY = r.nextInt(180) + 300;
            if (MO == 1) {
                    M_O_List.add(mO = new movingObstical(799, randY, GameSpeed));
            }
        }

        if (S_O_List.size() > 0) {
            for (int j = S_O_List.size() - 1; j >= 0; j--) {
                stationaryObject currSO = S_O_List.get(j);
                currSO.move();
                if (currSO.getXpos() <= -50)
                    S_O_List.remove(j);
            }
        }
        if (M_O_List.size() > 0) {
            for (int j = M_O_List.size() - 1; j >= 0; j--) {
                movingObstical currMO = M_O_List.get(j);
                currMO.move();
                if (currMO.getXpos() <= -50) {
                    M_O_List.remove(j);
                }
            }
        }
        //-------------------------------------end game scenarios -----------------
-----------------------------------------//
        collidingObjects();
        outOfBounds();
     }
     else{


     }
    }
    public void gameRender(){
        //testing in the creation of platforms
        //System.out.println("Its getting here though\n Height : " + height + "\n
Width : " + width);
        if(dbImage == null){
            dbImage = this.createImage(width,height);
            if(dbImage ==null){
                //System.out.println("Double buffered image is still null");
                return;
            }
            else{
                dbg = dbImage.getGraphics();
            }

            dbg.setColor(Color.black);
            Font fntGOM = new Font("ARIAL",Font.ITALIC,100);
            // loop the platforms so they keep on being drawn, just has to draw
each as they go
            //System.out.println("Makes it here");
               // System.out.println("The game is still trying ");
                dbg.setFont(fntGOM);
                dbg.drawString("You have lost OH NO",width/2,height/2);
        }else {
            if (!endGame) {
                dbg.clearRect(0, 0, 800, 600);
                for (int i = 0; i < pTest.length; i++) {
```

```
                    pTest[i].draw(dbg);
                }
                // dbg.drawRect();
                //System.out.println("ImageDB no longer equals null");
                //player printing
                if (S_O_List.size() >= 1) {
                    for (int j = S_O_List.size() - 1; j >= 0; j--) {
                        stationaryObject currSO = S_O_List.get(j);
                        currSO.draw(dbg);
                    }
                }
                if (M_O_List.size() >= 1) {
                    for (int j = M_O_List.size() - 1; j >= 0; j--) {
                        movingObstical currMO = M_O_List.get(j);
                        currMO.draw(dbg);
                    }
                }
                player.draw(dbg);
                //Draw the score and distance traveled
                dbg.setColor(Color.blue);
                dbg.fillRect(width - 200, 0, 175, 50);
                Font fntGO = new Font("ARIAL", Font.ITALIC, 15);
                dbg.setFont(fntGO);
                String dist = String.valueOf(distance);
                dbg.setColor(Color.black);
                dbg.drawString("Distance: " + dist, width - 150, 20);
            }else{
                player.draw(dbg);
                dbg.setColor(Color.black);
                Font fntGOM = new Font("ARIAL",Font.ITALIC,60);
                // loop the platforms so they keep on being drawn, just has to draw
each as they go
                dbg.setFont(fntGOM);
                dbg.drawString("oh no, you have lost!",0,height/2);
            }
        }

    }
    public void activePaint() {
        Graphics g;
        try{
            g=this.getGraphics();
            if((g!=null)&&(dbImage != null)) {
                g.drawImage(dbImage, 0, 0, null);
            }
            Toolkit.getDefaultToolkit().sync();
            g.dispose();
        }catch(Exception e){
           // System.out.println("graphics not working becasuse of error : " + e);
        }
    }
    public void setNewGameSpeed(){
        for(int so=S_O_List.size()-1;so>=0; so--) {
            stationaryObject currso = S_O_List.get(so);
            currso.setHmove(GameSpeed);
        }
        for(int so=M_O_List.size()-1;so>=0; so--){
            movingObstical currmo = M_O_List.get(so);
            currmo.setgSpeed(GameSpeed);
        }
    }
    //-----------------------------------game over conditions from object
collision-----------------------------------------------------//
    public void collidingObjects(){
        //set game over it
        if(collidingStationary()|| collidingMoving()){
            endGame = true;
            //System.out.println("the game should end here");
```

```
        }

    }

    //-----------------------------------end of game conditions-----------------
------------------------------//
    public boolean collidingStationary(){
        //check positions as they move using the lists
        for(int so=S_O_List.size()-1;so>=0; so--){
            stationaryObject currso = S_O_List.get(so);

            //top left of the current object
            int x = currso.getXpos();
            int y = currso.getYpos();
            //bottom right of the current object
            int xw = currso.getXpos() + currso.getWidth();
            int yh = currso.getYpos() + currso.getHeight();

            //top left of the player
            int plX = player.getXpos();
            int plY = player.getYpos();
            //bottom left of the player
           // int plXB = player.getXpos() + player.getHEIGHT();
            //int plYB = player.getYpos()
            // bottom right of the player
            int plXW = player.getXpos() + player.getWIDTH();
            int plYH = player.getYpos() + player.getHEIGHT();

            //touching the bottom of the player to the top of the stationary object
            if(plXW>=x && plX <= xw){
                if(plYH>= y){
                    //System.out.println("is hitting stationary object");
                        return true;
                }


            }


            //players right touching the left of the stationary object

            //player left touching the right of  the block.
        }
        return false;
    }
    public boolean collidingMoving(){

        for(int so=M_O_List.size()-1;so>=0; so--){
            movingObstical currmo = M_O_List.get(so);

            //top left of the current object
            int x = currmo.getXpos();
            int y = currmo.getYpos();
            //bottom right of the current object
            int xw = currmo.getXpos() + currmo.getWidth();
            int yh = currmo.getYpos() + currmo.getHeight();

            //top left of the player
            int plX = player.getXpos();
            int plY = player.getYpos();
            //bottom left of the player
            // int plXB = player.getXpos() + player.getHEIGHT();
            //int plYB = player.getYpos()
            // bottom right of the player
            int plXW = player.getXpos() + player.getWIDTH();
            int plYH = player.getYpos() + player.getHEIGHT();

            //touching the bottom of the player to the top of the stationary object
```

```
            if(xw>plX && x<plXW ){
               // System.out.println("The x pos is working");
                if(plYH>= y && plY<yh ){
                    return true;
                }

            }


            //players right touching the left of the stationary object

            //player left touching the right of  the block.
        }
        return false;


    }
    public void CurrentCollide(){
        // is player on the ground
        // for several other collisons
    player.setFlying(!onTheGround());
    }

    public boolean onTheGround(){
        int playXpos = player.getXpos();
        int playYpos = player.getYpos()+player.getHEIGHT();
        int playWidth = playXpos + player.getWIDTH();


        for(int i = 0; i<pTest.length;i++){
           int currplatXpos = pTest[i].getXpos();
           int currplatYpos = pTest[i].getYpos();
           int platWidth = currplatXpos + pTest[i].getWIDTH();


            if(playYpos >= currplatYpos){
                    //player iss on the ground as you increase further down you go
                if(playWidth >= currplatXpos && playXpos <= platWidth){
                   // if this returns true then they are on the ground.

                    // need to reset player pos
                    player.setYpos(pTest[i].getYpos()-player.getHEIGHT());
                    return true;
                }
           }//else if(playYpos < currplatYpos){
                //then the player is falling should activate the death sequence
somehow
            //}

        }
        // not touching any of the platforms
       // System.out.println("should not execute if the person is on the ground");
        return false;

    }
    public void outOfBounds(){
        if(player.getXpos() + player.getWIDTH() < 0 || player.getXpos()> 799 ){
            endGame = true;
        }
    }
}
```

movingObstical:

```
package objects;
```

```
import java.awt.*;

public class movingObstical {
    private int Xpos;
    private int Ypos;
    private int Hmove,Vmove;
    private int Height, Width;
    private int gSpeed;


    public movingObstical(int x, int y,int GameSpeed){
        this.Xpos = x;
        this.Ypos = y;
        this.gSpeed = GameSpeed;
        this.Hmove = 4;
        this.Height = 8;
        this.Width = 15;
    }
    public void move(){
        setXpos(getXpos() - gSpeed - getHmove() );


    }
    public void draw(Graphics g){
        g.setColor(Color.orange);
        g.fillRect(getXpos(),getYpos()-getHeight(),getWidth(),getHeight());
    }

    public int getXpos() {
        return Xpos;
    }

    public void setXpos(int xpos) {
        Xpos = xpos;
    }

    public int getYpos() {
        return Ypos;
    }

    public void setYpos(int ypos) {
        Ypos = ypos;
    }

    public int getHmove() {
        return Hmove;
    }

    public void setHmove(int hmove) {
        Hmove = hmove;
    }

    public int getVmove() {
        return Vmove;
    }

    public void setVmove(int vmove) {
        Vmove = vmove;
    }

    public int getHeight() {
        return Height;
    }

    public void setHeight(int height) {
        Height = height;
    }
```

```
    public int getWidth() {
        return Width;
    }

    public void setWidth(int width) {
        Width = width;
    }
    public void setgSpeed(int gspeed){
        this.gSpeed = gspeed;
    }

}
```

Platform:

```
package objects;
import window.scribbleFrame;
import java.awt.*;
import java.util.Random;

public class Platform {
    private int WIDTH,HEIGHT;
    private int Hspeed;
    private int Xpos,Ypos;
    private int repeats = 0;
    private Color PlatColor;
    private scribbleFrame sf;
    private Runner p;

    public Platform(int Xpos, int Ypos,Runner player){
        this.WIDTH = 50;
        this.HEIGHT = 20;
        this.Xpos = Xpos;
        this.Ypos = Ypos;
        this.p = player;
        setColor();
    }
    public void move(int Hspeed){
        this.Hspeed = Hspeed;
        //make size for box movemnets
        int Currxpos = Xpos;
        if(Collisions()) {
            setXpos(Currxpos -Hspeed);
        }else
        setXpos(799); //needs to be based on the previous platforms Y
    }
    //collide with player set player touch value to true or false.
    public int getWIDTH() {
        return WIDTH;
    }
    public void setWIDTH(int WIDTH) {
        this.WIDTH = WIDTH;
    }
    public int getHEIGHT() {
        return HEIGHT;
    }
    public void setHEIGHT(int HEIGHT) {
        this.HEIGHT = HEIGHT;
    }
    public void generateNewY( int prevYpos){
        Random r = new Random();
        int YposChange = r.nextInt(p.getHEIGHT()*2)+1;
        System.out.println("y pos change :" + YposChange);
    }
    public boolean Collisions(){
      // out of bounds
```

```
        if(getXpos() >= -49){
           return true;
        }
        return false;
    }
    // touching them players

    // not going to work so will just have to use the simpler method and use
seperate obstacles
//when it comes onto the screen set a new Ypos poision but can only be a certain
height apart at a time

    public void setColor(){
        Random rand = new Random();
        float r = rand.nextFloat();
        float g = rand.nextFloat();
        float b = rand.nextFloat();
        PlatColor = new Color(r,g,b);
    }
    public Color getColor(){
        return PlatColor;
    }
public void draw(Graphics g){
    g.setColor(getColor());
    g.fillRect(getXpos(), getYpos(), getWIDTH(), getHEIGHT());
}
public int getXpos(){
        return Xpos;
}
public void setXpos(int xpos){
    this.Xpos = xpos;
}
public  int getYpos(){
    return Ypos;
}
public void setYpos(int ypos){
    this.Ypos = ypos;
}
}
```

Runner:

```
package objects;

import java.awt.*;

public class Runner {

        //should lad on the nearest platform
        private int Xpos,Ypos,WIDTH,HEIGHT;
        private int Vspeed,Hspeed;
        private boolean Flying;
        private final int gravity = 9 ;
        private int prevPlayPos;
        private int jumpHeight;

        public Runner(int Xpos, int Ypos , int Vspeed, int Hspeed , int width, int
height){
            this.Xpos = Xpos;
            this.Ypos = Ypos;
            this.Vspeed = Vspeed;
            this.Hspeed = Hspeed;
            this.WIDTH = width;

            this.HEIGHT = height;
        }
```

```
        public void move(int CurrGameSpeed) {
            //need to make situations for Jumping, falling and running on the spot,
left and right
            // need to make the check for above or bellow the floor
            int currPlayPos = getYpos()+HEIGHT;
            // need to make it so that if Vspeed need to set a maximum height
            if(true){
                if (Flying) {// just above the line
                   //only resets on the ground or some kind of ground
                    // need to make it so it can jump above twice its height
                    // differene in position, jump height is the difference made
                    if (jumpHeight<-(getHEIGHT()*2.5)){
                        //need to get input from the user to determine speeds
                        setXpos(getXpos() + Hspeed - CurrGameSpeed);
                        setYpos(getYpos() + gravity);

                    }
                    else{
                        setXpos(getXpos() + Hspeed - CurrGameSpeed);
                        setYpos(getYpos() + Vspeed + gravity);
                       // System.out.println(" jumpHeight = "  + jumpHeight);
                        jumpHeight+=Vspeed + gravity;
                    }

                } else {// on the ground
                    setYpos(getYpos() + Vspeed);
                    setXpos(getXpos() + Hspeed - CurrGameSpeed);

                    jumpHeight = 0;
                }
            }
        }

public void draw(Graphics g){
    g.setColor(Color.CYAN);
    g.fillRect(getXpos(),getYpos(),getWIDTH(),getHEIGHT());


}
//positon and size

        public int getXpos() {
            return Xpos;
        }

        public void setXpos(int xpos) {
            Xpos = xpos;
        }

        public int getYpos() {
            return Ypos;
        }

        public void setYpos(int ypos) {
            Ypos = ypos;
        }

        public int getWIDTH() {
            return WIDTH;
        }

        public void setWIDTH(int WIDTH) {
            this.WIDTH = WIDTH;
        }

        public int getHEIGHT() {
```

```
            return HEIGHT;
        }

        public void setHEIGHT(int HEIGHT) {
            this.HEIGHT = HEIGHT;
        }


//speed and detection

        public int getVspeed() {
            return Vspeed;
        }

        public void setVspeed(int vspeed) {
            Vspeed = vspeed;
        }

        public int getHspeed() {
            return Hspeed;
        }

        public void setHspeed(int hspeed) {
            Hspeed = hspeed;
        }

        public void setFlying(boolean fly){this.Flying = fly;}

        public boolean getFlying(){
            return this.Flying;
        }


}
```

StationaryObject:

```
package objects;

import java.awt.*;
import java.util.Random;

public class stationaryObject {

    private int Xpos;
    private int Ypos;
    private int gSpeed,Vmove;
    private int Height, Width;

    /* Description
    * This object will be called on random platform but will not be called more
than twice in a row.
    * This object will simply act as a obsticle to jump over without dying. 1 hit =
death.
    * may need to change size as required could be built from various rectangles
    * Doesnt have to be baed on the platforms could be ArrayList
     *  */
    public stationaryObject(int xpos, int ypos,int gameSpeed){

        this.Xpos = xpos;

        this.gSpeed = gameSpeed;
//System.out.println("The X = " +getXpos()+ "and Y = "+  getYpos() +"for Stationary
object: " );

        Height = RandomHeight();
        this.Width = 20;
```

```
        this.Ypos = ypos - getHeight();
    }
    private int RandomHeight(){
        Random r = new Random();

        return r.nextInt(60)+30;

    }
    public void move(){

        setXpos(getXpos() - gSpeed);
  //      System.out.println("The X = " +getXpos()+ "and Y = "+  getYpos() +"for
Stationary object: " );

    }


    public void draw(Graphics g){
        g.setColor(Color.black);
        g.fillRect(getXpos(),getYpos(),getWidth(),getHeight());

    }

    public int getXpos() {
        return Xpos;
    }

    public void setXpos(int xpos) {
        Xpos = xpos;
    }

    public int getYpos() {
        return Ypos;
    }

    public void setYpos(int ypos) {
        Ypos = ypos;
    }

    public int getgameSpeed() {
        return gSpeed;
    }

    public void setHmove(int gameSpeed) {
        this.gSpeed = gameSpeed;
    }

    public int getVmove() {
        return Vmove;
    }

    public void setVmove(int vmove) {
        Vmove = vmove;
    }

    public int getHeight() {
        return Height;
    }

    public void setHeight(int height) {
        Height = height;
    }

    public int getWidth() {
        return Width;
    }

    public void setWidth(int width) {
```

```
        Width = width;
    }
}
```

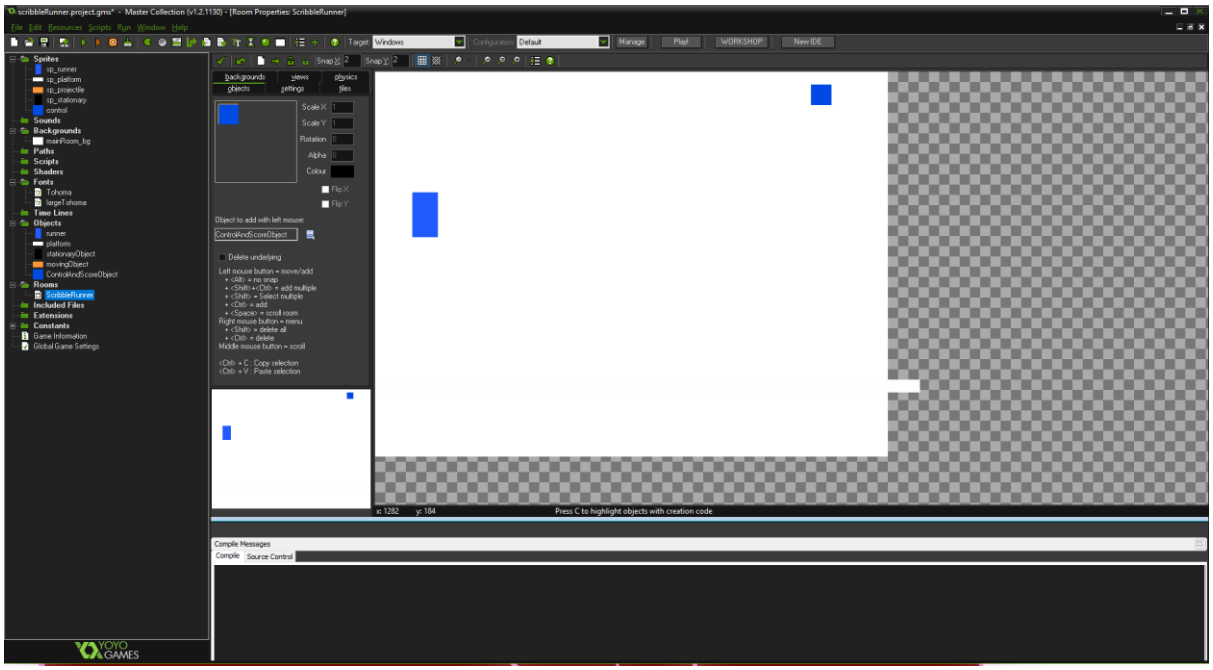
scribbleFrame:

```
package window;

import control.controlJoin;


import javax.swing.*;
import java.awt.*;

/**
 * Created by Matthew on 14/01/2018.
 */
public class scribbleFrame extends JFrame {
    private controlJoin panel;
    private String GameT = "Scribble Runner";
    private int WIDTH = 800;
    private int HEIGHT = 600;
    public scribbleFrame(){
     panel = new controlJoin(WIDTH,HEIGHT);
        //sS = new splashScreen(WIDTH,HEIGHT);


        this.setDefaultCloseOperation(this.EXIT_ON_CLOSE);
        this.setTitle(GameT);
        this.setPreferredSize(new Dimension(WIDTH,HEIGHT));
        this.add(panel);
        this.pack();
        this.setLocationRelativeTo(null);
        this.setVisible(true);
    }

    public int getWIDTH() {
        return WIDTH;
    }

    public void setWIDTH(int WIDTH) {
        this.WIDTH = WIDTH;
    }

    public int getHEIGHT() {
        return HEIGHT;
    }

    public void setHEIGHT(int HEIGHT) {
        this.HEIGHT = HEIGHT;
    }
}
```

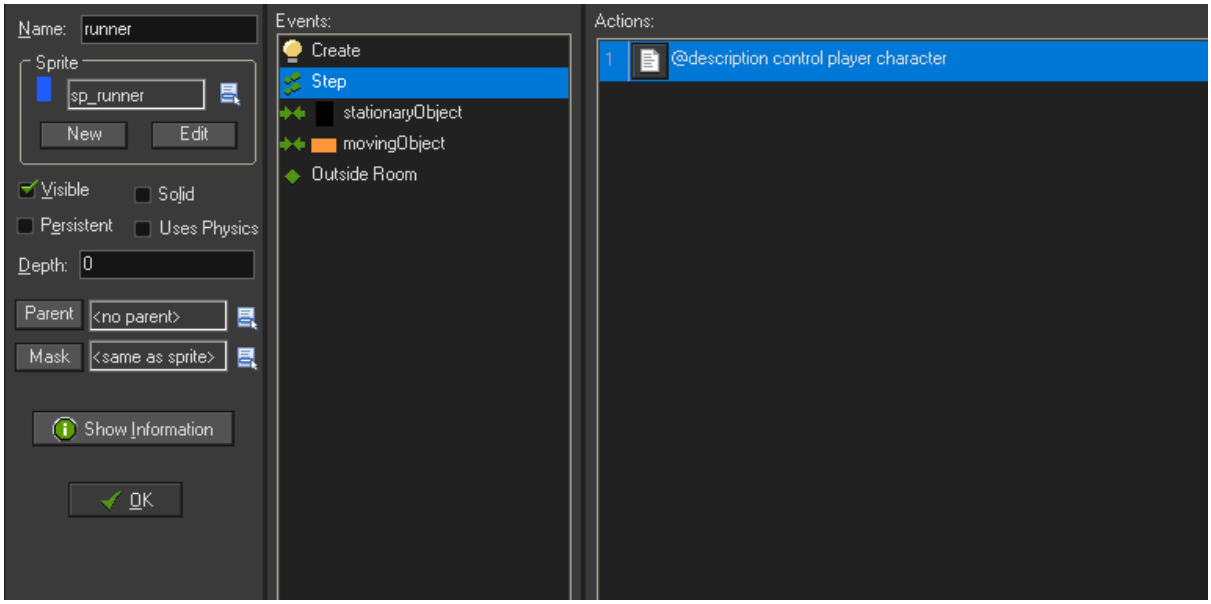
Appendix - B

Gama Maker development

The IDE:



Game Maker events:



Each Game maker object broken down in to events

Runner

Create event:

```
hSpeed = 0;
vSpeed = 0;
Gravity = 9;
Height = sprite_height;
Width = sprite_width;
jumpP = 0;
jumpHeight = sprite_height*2.5;
jumpB = false;
currYpos = 0;
maxJumpReached = false;
```

Step Event:

```
/// @description control player character
// You can write your code in this editor
//How can i make sure everything happens
// if runner off a surface make them accelerate/ move twords the surface
var Xpos = x;
var Ypos = y;
var RHeight = Height;
var RWidth = Width;
var hmove = 0;
var vmove = 0;
var jump=0;
var vPower = 0;
```

```
var jHeight;
if(!global.endGame){
if(keyboard_check_pressed(ord("D"))){
hSpeed = 9 + global.move;
}
if(keyboard_check_released(ord("D"))){
hSpeed = 0 ;
}
if(keyboard_check_pressed(ord("A"))){
hSpeed = -9;
}
if(keyboard_check_released(ord("A"))){
hSpeed = 0;
}
if(keyboard_check_pressed(ord("W"))){
jumpP = 15 + global.move;

}
if(keyboard_check_released(ord("W"))){
jumpP = 0;

}

if(Ypos+RHeight+Gravity>=platform.y && Ypos+RHeight<=platform.y){

//set player y  to platform Y and stop moveing dwon else keep moveing down
if(jumpB)
{
hmove = - jumpP;
}else{
Ypos = platform.y - RHeight;
show_debug_message("im running ");
hmove = 0;
// on the floor tehnivcally or touching a valid surface so
currYpos = Ypos;
jumpB = true;
maxJumpReached = false;
}
}else{
//just have to cap the jumping also on the floor
if(currYpos-Ypos >= jumpHeight){// ypos is changing all the time thats why its not working
maxJumpReached = true;
//ONLY WHEN ITS REASFY TO JUMP SHOULD IT DO SO, MAKE IT A BOOLEAN VALUE
}
if(!maxJumpReached){
hmove  = Gravity - jumpP;
jumpB = false;
}else{
hmove  = Gravity;
jumpB = false;
}
```

```
}
Xpos+=hSpeed;
Xpos-=global.move; //moving with the platforms
Ypos+=hmove;
x=Xpos;
y=Ypos;
}
```

Collison event Stationaryobject:

```
global.endGame = true;
```

Collision event movingObject :

```
global.endGame = true;
```

Outside room event:

```
global.endGame = true;
```

Platform

Create event:

```
/// @description Creatin of basic movemnent and resetting position
sprite_index = sp_platform;
randomize();
var R =(irandom(999));
randomize();
var B =(irandom(999));
randomize();
var G =(irandom(999));

image_blend= make_color_rgb(R,G,B);
```

Step event:

```
/// @description Insert description here
// You can write your code in this editor
var Xpos = x;

if(!global.endGame){
if(Xpos<=-49){
global.distance += sprite_width
 Xpos = 799;
}else{
   Xpos  -= global.move;
}
// gotta count the distance
//Creeat actual movement
x = Xpos;
}
```

stationaryObject

create event:

```
xpos = x;
SOypos  = y;
randomize();
```

```
var j = random_range(1,3);
image_yscale*=j;
```

Step event:

```
var ypos = SOypos - sprite_height;
if(!global.endGame){
x-= global.move;
y = ypos;
}
```

Outside room event:

```
instance_destroy();
```

movingObject

create event:

```
MOxpos=x;
MOypos=y ;
movespeed = 4;
```

Step event:

```
var ypos = MOypos - sprite_height;
if(!global.endGame){
xmove = movespeed + global.move;
x-= xmove;
y = ypos;
}
```

Outside room event:

```
instance_destroy();
```

ControlAndScoreObeject

Create event:

```
global.distance = 0;
global.move = 1;
global.endGame = false;
```

Step event:

```
randomize();
var i = round(random(120));
if(i==1){
instance_create(799,480,stationaryObject);
}
//show_debug_message("this is actually happeneing value of i : " + string(i));

var j = round(random(300));
var Ypos = round(random_range(300,480));
if(global.distance >=500 ){
```

```
if(j==1){
 //needs to spawn projecetile at a random y pos
   instance_create(799, Ypos,movingObject);
}
}
if(global.distance >= 500){
var i = global.distance/ 500;
global.move = i;
}
```

Draw event:

```
/// @description Insert description here
// You can write your code in this editor
draw_set_color(c_blue);
draw_rectangle(600,0,775,50,false);
draw_set_color(c_black);
draw_set_font(Tohoma);
draw_text(800 - 150, 20,"Distance: "+string(global.distance));
if(global.endGame){
draw_set_font(largeTohoma);
draw_text(0, 300,"oh no, you have lost");

}
```