Flex Project – Report

Valerie Gibson

James Wallace

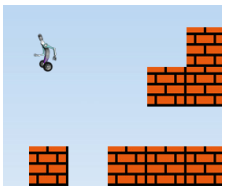
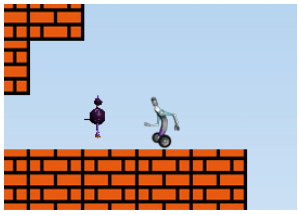
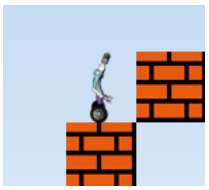
Contents

Part 1

Executive Summary

"Help our hero Thrustbot to make his escape from the land of the Wobblecats, who have all eaten too many curries and gone insane. Thrustbot must jump, fight and thrust his way out of danger to make his escape from the crazy Wobblecats. Curses!

As part of their maniacal plan, the Wobblecats have caused the sea levels to rise. Thrustbot must make his escape without being bounced into oblivion by a Wobblecat or drowned in the sea."

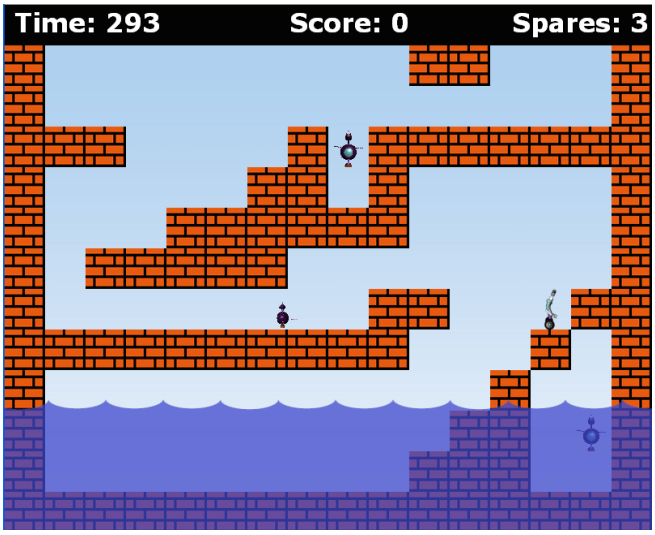




Thrustbot vs Wobblecat is a 2D platformer game written using Adobe Flex and ActionScript, executed on the Flash Player. It was a pair project between the two student developers, Valerie Gibson and James Wallace, for the 3d Modelling and Animation module coursework.

Designed in the old style of everyone's favourite platform games like Mario and Sonic, you will soon recapture the magic of those nostalgic moments spent playing on the MegaDrive and SNES in the early 90s.

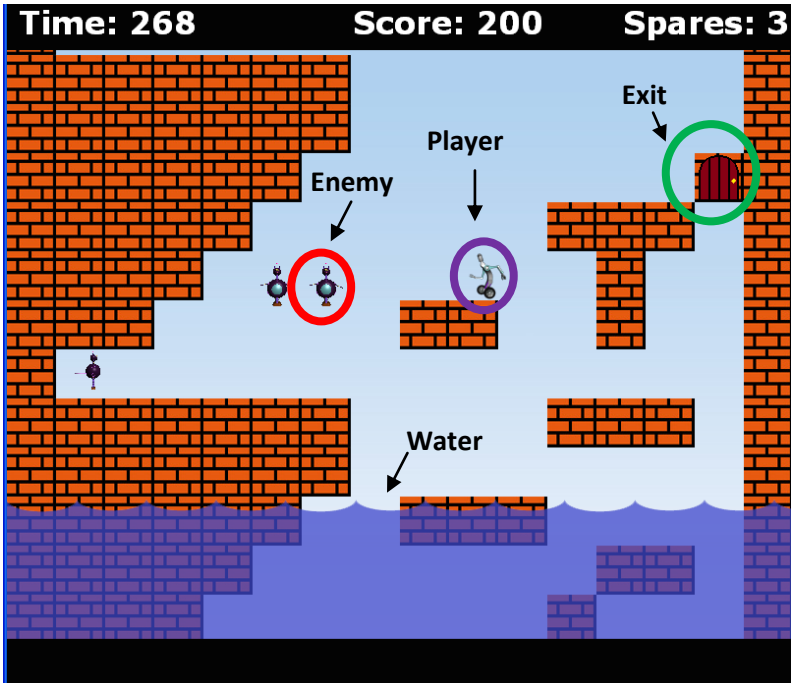
Back in those days consoles were bulky and cumbersome, with games that had high price tags and long development times. Now the games available are only limited by your imagination and taste, not by the size of your bedroom or wallet.

Enjoy Thrustbot and his adventures anywhere you have access to the web! Throw away vector graphics and complicated physics engines: reignite your love for the old style platformer game and help Thrustbot escape!



Game Objectives

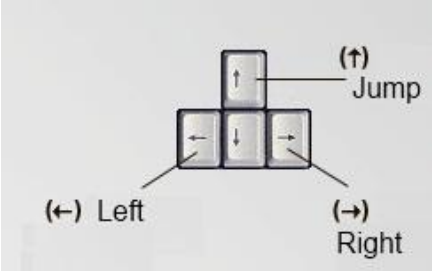
The objective of the game is to help Thrustbot make it to the exit at the top of the map. Thrustbot must not run into enemies, or fall in the water.



Game Play

Controls

The player controls Thrustbot using the arrow keys.



Thrustbot does have an element of speed and friction, so the player must be careful when controlling him! Tapping the keys makes it easier to handle than holding them down. Controls have been made deliberately trick to increase the difficulty of the game.

Thrustbot may only jump if he is on the ground; he may change direction mid air but he cannot jump.

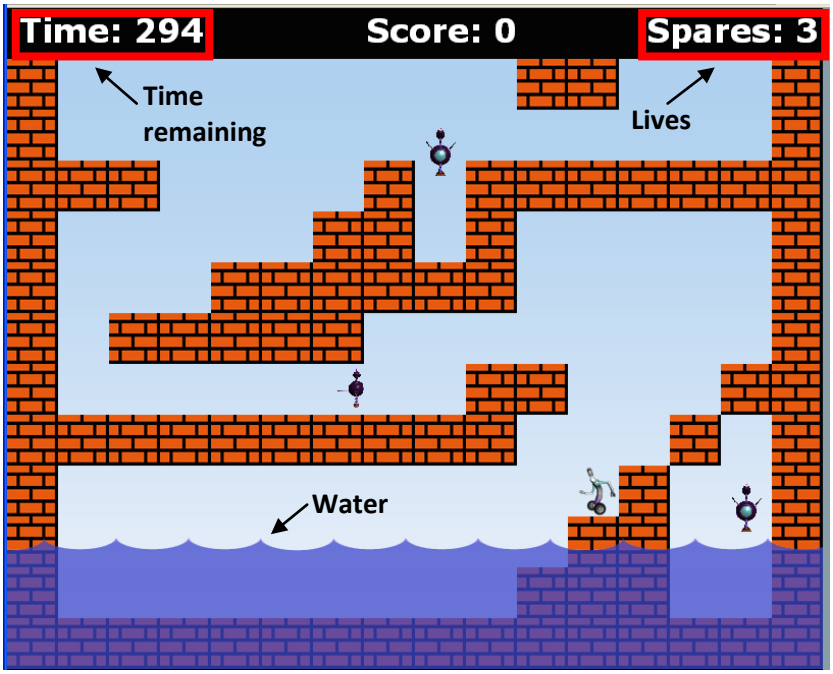
Scoring Points

The player scores points by making it to the exit of the level. Each level has a timer which counts down to zero; the time it takes to make it to the end is deducted from the time remaining for the map to give the score.

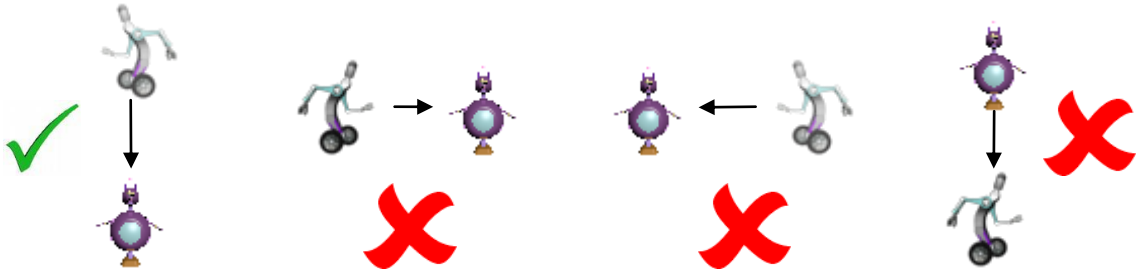
Bonus points will be awarded for completing the game on more difficult settings, as well as disabling Wobblecats and keeping lives intact. When wobblecats die, they turn into a big head – bonus points are awarded for clicking it with the mouse before it disappears.

Game Over

The game will be over when the player has no lives left, falls in the water or the timer for the map runs out.

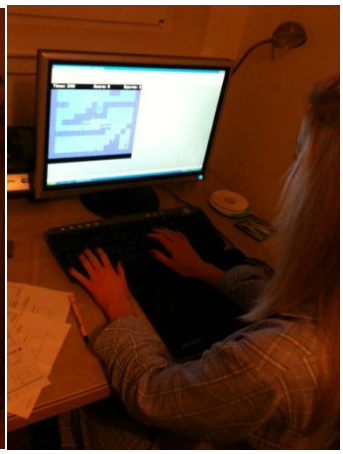
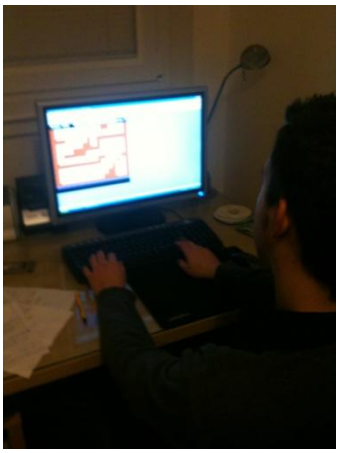


Thrustbot carries around a limited number of spare parts which he can use if he encounters a Wobblecat and fails to defeat it. This gives the player 3 lives. Lives are deducted every time Thrustbot runs into an enemy. Lives will not be deducted if Thrustbot hits them on the top of the head from above.



Game Development

The game was developed using 3 main iterations of prototype, with user feedback at each stage. 2 testers were involved from the first iteration who were selected due to their widely varying programming and computer knowledge. Tester 1 is a full time SQL developer who is very computer literate. Tester 2 is an accountant with very little knowledge of computers. This allowed us to compare the feedback of these 2 differening types of users.

We had an initial brainstorming session where we drew some sketches and came up with ideas for the game. There were a few possible approaches we considered – ranging from a small game where the user throws balls at running enemies to a small platformer on one static screen.

Iteration 1

In iteration 1, we made a very simple one-screen game with a few platforms and no animation on the player or enemies. There were only tiles and a ball which acted as the player and enemy for demonstration purposes. We did not implement any collision detection or player controls such as jumping or running.

We came to some design decisions from our own opinions and the views of the testers. The level would too easy to complete, so we had to find ways to make it more interesting.

Main requirements for the next iteration were:

Task	Main Area	Priority
Key controls	Player	High
Friction	Player	Medium
Add Momentum	Player	Medium
Collision detection top/side hit	World/Enemy	Medium
Add water hazard	World /Enemy	High
Add scroll	World	High

Iteration 2

In this iteration, we implemented a scroll mechanic with a way to create maps very easily. This allowed us to change the level easily. We decided to use Wobblecat as the enemy, since his bounce animation fitted it well. We made two versions of enemy – a patrolling one that walks from side to side and one which jumps up and down.

To make the game more interesting and challenging we added the water effect that chases the player up the screen. We added friction and movement to the player.

We did not implement any animations on the enemies or players. We used very basic pictures of the letters "L", "R" and "U" to determine whether the object was facing left, right or jumping upwards respectively. This allowed us to implement the mechanics for the jumps, patrols and movements without actually animating them.

The testers were pleased with these changes, but asked if we could make the controls easier to use which we did – still leaving them tricky enough to make it interesting.

Requirements for last iteration were:

Task	Main Area	Priority
Refine jump mechanic	Player	High
Add animations	Player	High
Add Win Game	World	Medium
Make collision detection more efficient.	World	Low
Add player lives	World/Player	Medium
Add difficulty setting	World/All	High
Add animations	Enemy	High
Add scoring	Application/World	High
Add save score	Application	Medium
Add Game Over	World	High
Smoother scrolling	World	Medium
Add Start screen	World	High
Help function	World	High
Add player death	Player	Medium

Iteration 3

The last iteration was used to polish the game mechanics and fix any bugs.

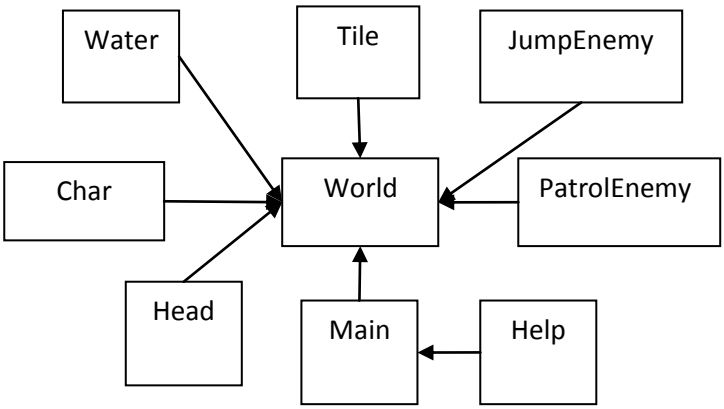
We added collision detection using a point-based system. We used 4 points – one on the left, right top and bottom to detect a hit from that direction. All animations were added to the player and enemies, including one for when the player dies.

All remaining requirements from the coursework project specification were added as well as those identified in the last iteration. This included the database access; high score display and recording; help screen and difficulty levels.

To add another dimension to the game, we added a floating head which appears when an enemy is killed. Players can gain bonus points by clicking it in time before it disappears. The testers thought this was a very novel idea and good fun.

Overall Design

There are 8 main modules in the application:



1. Main

This module is used as an initial splash screen which is shown to the user. It handles:

- Displaying the high scores
- The option to view the help screen
- Starting the game
- Setting the difficulty level

This module communicates with the Help module to display the help screen. It also communicates with World to send it the difficulty setting chosen by the user.

2. World

This module is used as a central point of communication for all the other modules. It is responsible for a lot of the game functionality at the top level. It handles:

- The drawing and updating of the map
- Scrolling
- Collision detection between player and objects
- Player life count
- Game Over and winning the game
- Player movement

This module communicates with all the other modules, except Help.

- Instantiating a MapBuilder class which then returns a matrix map used to generate the level in the game.
- Telling Water to initialise when the game is started.
- Altering Char's friction and Water's advancement speed depending on the difficulty level
- Makes a call to the Char module to move the player when a key press is detected

- Monitoring the state of collisions between enemies, the player and the environment every frame. This involves calls to the player and enemy x and y coordinates.
- Notifying the Char module when the player has died
- Initialising the initial x and y positions of Tile, JumpEnemy and PatrolEnemy when they are added to the map
- Setting the velocity of the JumpEnemy
- Accessing the image source of tiles when they need to change when a scroll happens
- Setting the walkable property of tiles when the map scrolls
- Telling the Char whether it can jump or not
- Sending the final score of the player back to the Main module to be stored in the database
- Notifying the Main module when a restart is required or the game is over / has been won

3. Char

This module is responsible for the characters motion, animation and properties. It handles:

- Character speed
- Character friction
- Animation of the characters behaviour (walk, jump or die)

It does not communicate with any other module, except calls from World to itself.

4. Water

This module is responsible for the water effect. It handles the speed the water advances and the advancement of the water up the screen every frame. It does not communicate with any other module, but receives calls from World which it responds to.

5. Tile

This module represents one tile on the map i.e. one 50*50 square on the screen. It handles the scroll function when all the tiles must be altered when the player moves up the screen. It does not communicate with any other module, only responding to the World when called on to do so.

6. PatrolEnemy

This module represents a patrolling enemy, which walks from side to side. It handles:

- Enemy movement speed
- Maximum steps taken by the enemy
- The animation of the patrol

It does not communicate with any other module, except with responses to World.

7. JumpEnemy

This module represents a jumping enemy, which jumps up and down. It handles:

- Enemy jump animation

- Maximum height of the jump
- The speed at which the enemy jumps
- The animation of the jump

It does not communicate with any other module, but receives calls from World.

8. MapBuilder

This is a small helper class that returns the relevant map for a particular difficulty level to World.

9. Help

Displays a splash screen which shows helpful information like how to play the game and score points. It does not communicate with any other class, and may only be instantiated by Main.

10. Head

The Head module is used to create a floating head that bounces around the screen when an enemy is killed. The player will receive bonus points if they click with the mouse in time before it disappears. The module is responsible for:

- Setting the velocity and direction of the head
- Setting the duration the head appears for
- Determining when the head has been clicked

Joint Conclusions

Best Parts of the Game

The best parts of the game are:

- Platformer-based design
- The water effect
- The scrolling mechanism
- The player and enemy animations

Implementing the map-based platformer idea was difficult since it involved many mechanisms to make it work. This included the scroll mechanism and a lot of collision detection. We had to be really careful when we were designing the code architecture around this, since too many detections going on would eat memory and cause all sorts of problems.

The scrolling mechanism helped us to expand the size and complexity of the game, and took a while to figure out. Many tutorials are writing based on the Flash IDE and it was impossible to use the material directly. Once we had the general idea, it was up to us to actually implement a solution in Flex.

Playability

Overall we are very happy with the game. It meets the purpose we set out to achieve – a fun little platformer in the style of Mario.

The water effect is a really nice way off adding an extra level of complexity to the game. Our testers really liked it and agreed it was a nice way of making the game more interesting.

The player and enemy animations are taken from 3ds max and add a really nice touch to the game. It is great to see our models represented in this way and being manipulated in the game. The enemy death animation was added in at the end and we are really happy with it.

The controls could be handled better and this was touched on by our testers. We could not find a way to get smooth motion, since the key events seemed to block each other – thus making it impossible to get a smooth motion. In the end however it was felt the controls were suitable for the purpose and made the game tricky, which was good.

At the last iteration our testers seemed very pleased with what we had done. Tester 1 said they found it quite challenging, and Tester 2 said it was not overly complex but difficult enough to be interesting “in that frustrating way like Mario”.

Project Organisation

While developing the game, we worked on a task list with designated to-do items for each developer. These were broken down into high, medium and low priorities. We sorted these into a per-module basis and this allowed us to work individually but keep track of overall progress. Snapshots of these were shown in the Game Development section.

We used SVN to coordinate our source code. This allowed us to keep track of each other's progress and fit modules together easily. We had weekly meetings about the project, and discussed the current progress and next steps. These meetings were also used to help each other solve any particularly complex problems we were stuck on.

Possible Improvements

The game could be expanded to use different images for different maps. More objects could be added to the environment for the player to interact with – like in Mario where there are power ups and extra lives. Music could be added to the game to add atmosphere, as well as some sound effects for when the player dies etc.

Evaluation of Software

Flash Player

The flash player offers great applications and rapid development, but lacks access to specific functions and control of the system that can be found in traditional JavaScript or web applications. The security sandbox is perhaps too tight.

Flex Builder IDE

As Flex Builder is based on Eclipse, it is really easy for an Eclipse user to pick up and start developing applications quickly. It is good to have a familiar environment to develop in. However, the drag and drop components are a wonderful addition which is not standard in eclipse and makes developing 78GUI applications much easier. Auto-completion of mxml is helpful as well as having access to the mxml and ActionScript apis and intellisense.

However, the apis and compiler still need work – often there may be errors which are not detected until runtime, or are not detected at all One can spend a long time looking for an error, only to find it is something as simple as a missing colon.

It was great working with Flex and ActionScript, though it would have been great to see its full potential in the development of RIA applications rather than animation. It was enjoyable nonetheless.

3ds Max

This is an excellent piece of software with lots of tools and functionality - far too many to learn in a short time. It is daunting when approaching it at first, since it is a vast and complex tool. It is very much focused on artistic modelling unlike the CAD tools which are very much orientated to representing the real objects rather than making them look pretty.

This was a real joy to learn, after getting over the first stages of understanding the basics and what it is actually capable of. It is the perfect embodiment of the "a fool with a tool is still a fool" situation – it is such a highly complex and professional tool and we could only produce very fundamental objects like cubes etc.

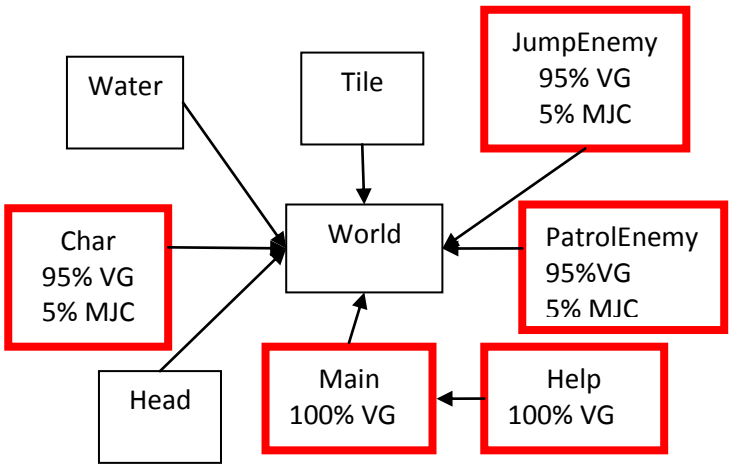
Part 2 – Individual Report

Valerie Gibson – 4th year MEng Computer Science

Summary of Role

I was primarily responsible for implementing the Main, Help, Char and PatrolEnemy and JumpEnemy modules. This involved many aspects of programming – from animating the still images into the behaviours of the characters to implementing the initial splash screen and GUI components. I was also responsible for generating the still images from 3ds max.

I feel I added value to the team by performing well in all of these diverse programming tasks assigned to me as well as writing my share of the report. I spent many late nights working on the project and dedicated a lot of time to it. I feel I have really put a lot of effort into the project, both for this part and the modelling side of things.



Module Descriptions

I tried as far as possible to modify or improve the code to make it my own, helping to further my own understanding of the content. Since the game is quite different from the course examples there was a lot of scope for novel or innovative use of the course content.

1. Main

This module is the first screen that is shown to the user. It gives the option to view the help section, start a new game or look at the high scores. It retrieves all of the scores from the database using a modification of the PHP script we were provided with – removing the need to use two scripts.

Important Functions:

startGame(): starts the game, passing the difficulty chosen by the user to the World module which is instantiated. The World is added to the display and the focus is set to it.

restart() and restartGameOver() are called by World when the player chooses to reset the game upon death or winning the game. This is sent back to Main where the game is then restarted. This then retrieves the newly inserted score along with the other results from the database and displays it on the screen.

Evolution

This module was not implemented until the final iteration of the application. It was a little tricky to convert the World module into its own class, separate from the application since it changed many of the local calls to that of the parent. However, this reduced the complexity and time spent on it in the early days which was spent doing more important tasks at the time. This module started off as a simple "Start" button which began the game. The difficulty levels were then added followed by the high score board and link to the help section.

2. Help

This module shows some helpful information to the user such as the game controls and objectives. It is invoked by World.

This module was implemented last of all since it was relatively easy to make and simple to connect to the rest of the application.

3. Char

This module represents the character played by the user. It handles character movement speed, friction and the animation of the characters behaviour. The character can walk left and right, jump and also die.

I would like to highlight that although this is my own module, I worked with James when doing the character motion.

Important Properties

Public variables vy, ay, vx and ax control the velocity and acceleration of the player. These were taken from the course content. These are accessed by World at various times including during collision detections and difficulty level setup. Public variable friction is also accessed in a similar way.

Private variable imgArray contains the images for player walk animation, with the player facing right; private variable imgArrayLeft are the same images as array above, but flipped horizontally and private variable imgArrayDeath holds the images for the death animation.

Important Functions

movePlayer() is responsible for the movement of the player across the map. This is called by World on detecting a left or right arrow key press.

jump() is simply that – it allows the player to jump. This is invoked by a call from World upon detecting an up arrow key press. If the player is on the ground they may jump, or walk right and left. However, if the player is in the air they may not then jump again – they may only change direction.

die() performs a death animation when the player is runs out of lives, falls in the water or the timer runs out.

advance() handles the enter frame function and the animation of the Char. This is invoked by World. The animation loops over all the frames of the relevant animation. If the player is walking, then the animation will loop through the imgArray until it reaches then end, then play the animation frames in reverse.

This function also handles the player motion. It stops the player going above the maximum velocity specified and progresses their movement to where they should be for the next frame to be shown. This involves some of the code from the course content.

Evolution

The Char was originally represented by the infamous Blue Ball from the lecture examples. Directions were shown by adding an "L" or "R" to indicate if the player was facing right or left. This meant that the programming of motion could be done without worrying about the actual animations.

The proper animations were added in the second iteration, using stills from the 3ds max model files. The friction and jump mechanics were also added at this time. In the third iteration the hit tests were made more efficient by using the four-point system mentioned previously.

4. PatrolEnemy

Patrolling enemies walk from side to side on the screen. This module dictates their movement speed, number of steps taken from side to side and the animation of the patrol.

Important Properties

Public variable bottom is used to detect when the enemy has fallen off the bottom of the screen and needs to be removed. This is accessed by World when advancing the map when the player scrolls, and updating the screen.

Private variable imgArray and private variable imgArrayLeft again hold the images for the patrol animation, facing right and facing left respectively.

Important Functions

init() sets up the arrays with all the images used in the animation. It also adds the on enter frame event.

doPatrol() is the call-back for the on enter frame event. This cycles through the still frame images until the last one is reached, then plays it back in reverse. It also determines the position of the enemy on the screen.

scroll() stops the enemy moving up with the screen when the player invokes a scrolling of the screen. This is called by World.

Evolution

The enemy started life as a stationary Blue Ball then in the second iteration the animations were added from 3ds max. The jump motion was also added in this iteration.

5. JumpEnemy

Jumping enemies jump up and down between the ground and a certain height. This module is responsible for the enemy jump animation, the speed the enemy jump, the animation of the jump and the maximum height the enemy can jump to.

Important Properties

Public variable bottom is used to detect when the enemy has fallen off the bottom of the screen and needs to be removed.

Private variable jumpArray holds the images used in the animation of the jump. There is no need to reverse them, since the animation is played in reverse at the height of the jump.

Important Functions

init() adds all the images to the array and sets event listener for the on enter frame function.

setInitVy() is used to set the initial velocity of the enemy. This is called by World at initialisation.

doJump() cycles through the jump animation for the enemy. Once the maximum frames have been reached, the animation is played in reverse. This also handles the movement of the enemy, up and down and where it will be placed on the screen. This is based on some of the code from the course.

scroll() stops the enemy from moving up with the screen as the player advances.

Evolution

First the motions for the enemy were programmed using a basic image which did not change. Then in the last iteration of the program the actual animations were added from the 3ds Max stills.

What I Have Gained

I have learned a lot about 2D animation and how ActionScript and Flex can be used to make Flash style games. I gained a great insight into how Flex works and how ActionScript can be used either from Flex or Flash to make similar kinds of games. Although Flex may not be directly aimed at this purpose, I feel I learned a lot from the project about the capabilities of the language and about 2D animation as a whole.

The project pushed my time management and mental stamina to the limit at times, since this terms workload has been one of the hardest challenges I have faced during university. In this respect, I feel I grew a little stronger than I was before at pushing myself to work hard and motivate myself.

The project also helped me to gain confidence in my programming skills. I have a very delicate confidence regarding my own capabilities and I often feel I am not as gifted as many of my peers. However, though working hard on this project I proved to myself that maybe I am not so hopeless and I often feel!

The course could be improved by making the coursework requirements clearer – especially for the report. The marking scheme and project specification did not seem to match up. To get into the highest bracket one has to do a lot of work for a relatively small application. This is fine, as long as what is expected is made clear at the beginning. The requirements for the game could also have been a bit clearer, since many people seemed to believe many different things.

Another improvement would be if the coursework for each section of the course were better coordinated. They are both extremely large pieces of work and it was unclear that we were supposed to have finished this part before the second half was given. This lead to a lot of stress!

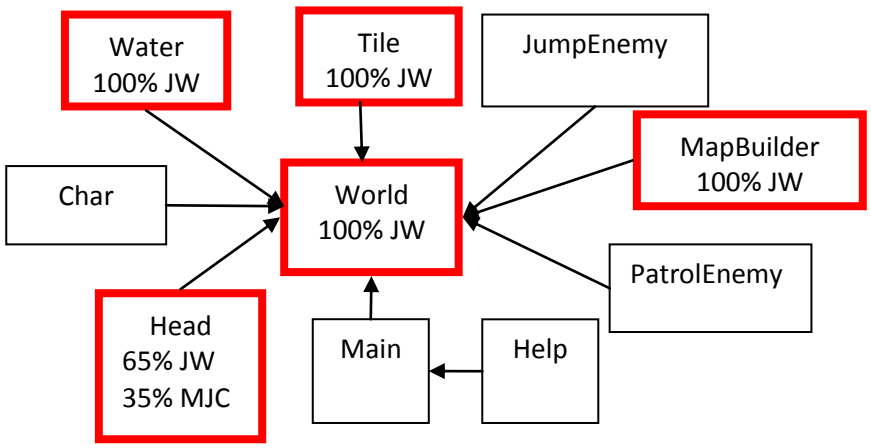
Part 3 – Individual Report

James Wallace – 4th Year MEng Computer Science

Summary of Role

I programmed the World, Head, Tile, Water and MapBuilder modules. This involved many programming challenges I had to overcome. This included finding a way to add the water effect without using a lot of memory; implementing the scrolling function and many hit test detections which were central to the game play.

I contributed to the project by investigating the way the main game concept would actually work – how to make it scroll, how to layout the maps and I undertook a lot of research into tile based games. These were mainly Flash based tutorials. I also did a lot of research into how games like Mario actually work.



Module Descriptions

1. World

The World is the top level controller for all the other modules. It controls many of the individual interactions of the other modules. This includes updating the positions of enemies and the player, detecting collisions, keeping count of the player's lives, detecting the game end and handling the scrolling function.

I would like to highlight that although this class is attributed to me, I did cooperate extensively with Valerie when bringing all the modules together and doing the top level interactions.

Important Properties

Public variable difficulty is written to by Main and sets the difficulty level of the game. This can be either "easy", "medium" or "hard" as offered to the user in the ComboBox on the Main screen.

Private variable currentLives determines how many lives the user has. Private variable time stores the time it takes the user to complete the game. Private variable score holds the players current score.

Important Functions

startApp() initialises the games and sets up the parameters for the chosen difficulty level. It instantiates the module MapBuilder which then returns a matrix map used to generate the level in the game.

keyHandler() is an event handler which detects keyboard inputs from the user. This can either be the up, left or right arrows. These key inputs are sent to the Char module which then moves the player on the screen accordingly.

nextFrame() is the event callback that is used to handle each enter frame event. It communicates with the Char module to tell the player to move to the relevant position that it will be displayed in the next frame. This method detects if the game is over and if so tells the Char module that the player has to do the die animation.

buildMap() uses the matrix returned by MapBuilder and constructs the items which are displayed on the screen initially – this is the first 12 rows high and 16 columns wide. These are all Tile types, from the Tile module.

addMapRow() handles updating the contents of the screen after a scroll; when the screen scrolls down more than 50px a new row of tiles must be added to the top of the screen. It interfaces with JumpEnemy, PatrolEnemy and Char to set their initial conditions.

scroll() allows the player to progress upwards through the game while occupying the same screen area. This is achieved by moving the map down whenever the player gets above a certain height on the screen. The module must communicate with the Water, Char, JumpEnemy and PatrolEnemy.

checkCollisionAndRefresh() is the environment watcher function, which checks for collisions between the children of the canvas. This includes the player, enemies and the environment such as the tiles and the exit door and requires communication with them accordingly to move them into the relative positions.

A four-point collision test was implemented on top of the default Flex collision hit test. This allowed for the detection of the direction that the object got hit, which was important for deducting lives or adding bonus points.

Evolution

This module grew in size and complexity and new modules were added to connect to it and more functionality was added. The first iteration allowed a static map to be drawn on the screen and a basic player to be added. The second iteration had basic hit tests and collisions, as well as scrolling and a map matrix. The final iteration had more robust hit tests and access to the MapBuilder for different maps for different difficulty levels as well as player lives and all the other extra functionality we added to meet requirements from testers and the project specification.

2. Head

This module allows a head to appear when an enemy is successfully killed by a player. The head spawns at the location of the dead enemy and takes on a random velocity in both x and y. Every 10 frames the velocity is modified via a random number creating a random effect making it hard to click.

The code for the bounce mechanic is from the course content. The Head module sends an event back to the World which tells it to increase the players score on mouse click. There is a second event which is sent to tell the World to delete it when it has reached the maximum number of bounces before it should be removed.

Important Variables

Private variable bounceCount keeps track of how many bounces the head has already done.

Important Functions

doBounce() is the call-back for the on enter frame function. It handles movement of the head and boundary conditions like hitting walls. It is also in charge of randomly changing velocity.

mouseHandler() is the call-back for the mouse click event. It lets the World know that the player has successfully clicked it.

3. Tile

This module represents one tile on the map. Each tile is one 50*50 square which can either be a platform that the player can run on, but not through, sky, an enemy or the exit.

Important Variables

Public variable walkable is a Boolean value which determines whether the player can walk through the tile or not. This is very important for the collision detections in the World module, as well as setting up the map and updating the screen.

Important Functions

scroll() adjusts the position of the tile when the player moves upwards. Tiles must not move up with the player, they must stay fixed or fall off the screen and be replaced with new ones from the map matrix dictates.

Evolution

The Tile module appeared in the second iteration for the implementation of the scroll function and did not really change much from then.

4. Water

This module is responsible for generating the effect of the water advancing up the screen.

Important Functions

init() is called by World when the game is being initialised. It adds the on enter frame event to the module.

scroll() adjusts the position of the water when the player move sup the screen. It is called by World.

Evolution

At first a plain blue image was overlaid on the screen, in the second iteration of the application. In the final version I added the wave effect and made it look better.

5. MapBuilder

This ActionScript class is used to return the appropriate map to the World module. This stops the maps cluttering the screen in the main file. A corresponding map will be returned depending on whether the user has chosen Easy, Medium or Hard from the Main splash screen ComboBox dropdown.

This module did not exist until the last iteration, where it was felt that the 3 different difficulty maps were cluttering up the screen too much.

What I Have Gained

I learned a lot about Flex and ActionScript and how it can be used in animation. I improved my programming skills and general animation knowledge. I worked very hard on the game and my personal management skills were improved. I also learned about what Flex is actually capable of, and the directions that this kind of RIA framework is taking.

The course could be improved by splitting the course into two distinct modules, since the two parts are very different. The Flex coursework would be better as an individual project – separating the code percentages and one author per module was extremely difficult.