LOGIC WARS

Design Portfolio by Luke Geeson (lxg03u) and Benjamin Middleton (brm03u)

Words: 3020

Game Concept

We aim to emulate an agile development team working for a small game development company, called NG2 Games. We are required to produce an inventive 2D game, including a prototype and larger full version and all supporting documentation. The game aims to emulate a typical 'space invaders' game crossed with a 'shoot 'em up' such that the enemy will continuously fire randomly and aggressively. The game adopts well known concepts from other similar arcade games such that it satisfies the requirements extracted from the user stories.

Game Requirements

User Stories

User stories were used in the requirements process. It enables us to recognise specific functions and specifications to cater for a variety of users from the ‘average person’ to the ‘hardcore gamer.’ The user stories were used as a basis for the requirements of the product we would create.

The Casual Gamer

The casual gamer is someone who plays the games more for self enjoyment and finds little to some time in the day to wind down playing games they can get instant fun from.

- As a casual gamer, I want to be able to control the spaceship with arrow keys and "WASD" like I could in any other game so that I could pick it up quickly and/or cater to the playstyle I prefer.
- As a casual gamer I want minimalist controls in game and on the menu so that it isn’t a bother to pick up and play the game in a quick and easy fashion.
- As a casual gamer, I want cool power ups, providing temporary or lasting boosts so that you have "an edge" in game and providers further depth of play.
- As a casual gamer, I want some passive instructions which provide useful tips so I can play the game easier and progress faster e.g a control screen so that I may learn the controls.
- As a casual gamer, I want to be able to put down the game as easily as I can pick up e.g. a simple pause function so that I may return to it later.
- As a casual gamer, I want to be able to restart the game should I lose so that I may try and beat my score.

The Average Gamer

The average gamer is someone who finds playing video games a hobby, something that they can spend a considerable duration of their weekends immersing themselves in yet still finding time to cater for their other hobbies.

- As an average gamer, I want to monitor my score and be able to compare against friends for a social, friendly, competitive edge.

- As an average gamer, I want a challenge in the form of a sophisticated A.I which have a varying degree of difficulty based on my progression in the game.
- As an average gamer, I want a lasting experience where I can play the game continuously through the means of additional lives or a checkpoint system.
- As an average gamer, I want each session to be fresh and new, providing a fair level playing field each time I play the game.

The Hardcore Gamer

The hardcore gamer is someone who plays games on a very competitive or serious level. They immerse a substantial amount of their free time playing video games and dedicating themselves to ranked progression for a sense of self-confidence and/or feel for achievement.

- As a hardcore gamer, I want to be able to benchmark the games performance compared to my computer system in order to get the highest quality of graphical satisfaction.
- As a hardcore gamer, I want to not only compare my scores against my friends, but compare my score to other players through a ranked scoreboard
- As a hardcore gamer, I want to have a feed which keeps me up to date on news related to the game which is easily accessible and requires minimal effort
- As a hardcore gamer, I want to be able to strategise and find 'min-max' methods to achieve the highest scores possible
- As a hardcore gamer, I want secondary objectives that provide further ranking up e.g. an achievement system that I can also compare amongst friends and other players.

Requirements Specification

This section denotes the requirements specifications of the product with prioritisation according to necessity.

Functional Specifications

the functional specifications are those which must be implemented in the game to enable basic functionality of the requirements.

Game Window

1.0	The game will have a window resolution of 600px by 400px	**Mandatory**
1.1	The game window will be labelled 'LOGIC WARS'	**Desirable**

Player

2.0	The player will be a playable spaceship shaped like a logic gate	**Mandatory**
2.0.1	The player's ship will have a detection box the same size as the player sprite (33x49px).	**Mandatory**

2.0.2	Player movements must be tracked and mapped to ship animations (changing images) to indicate movement.	**Desirable**
2.1	The player sprite will face towards the top of the screen.	**Mandatory**
2.2	The player's ship will have a designated playable area at the bottom of the screen (y in the range of 350 - 575)	**Mandatory**
2.3	The player's ship will be controlled via inputs on the keyboard	**Mandatory**
2.3.1	The player's ship will move with 'wasd' or the arrow keys	**Mandatory**
2.3.2	The player will fire projectiles from the ship with the space bar key.	**Mandatory**
2.3.3	The player will be able to access the pause menu with the 'p' key	**Desirable**
2.4	The player will be able to fire a weapon from the player sprite	**Mandatory**
2.5	The player will have a health value	**Mandatory**
2.6	The player will have a live(s) value	**Mandatory**
2.7	The player will have a score value	**Mandatory**

Enemies

3.0	The enemies will be spaceships shaped like negated logic gates.	**Mandatory**
3.0.1	The enemies' ships will have a detection box the same size as the enemy sprite (33x49px).	**Mandatory**
3.0.2	enemy movement must be tracked and mapped to ship animations (changing images) to indicate movement.	**Desirable**
3.1	The enemy sprites will face downwards on the screen	**Mandatory**
3.2	The enemy ships will navigate towards the bottom of the screen from their origin at the top of the screen.	**Mandatory**
3.2.1	The enemy ships will be removed upon destruction or when they leave the screen.	**Mandatory**
3.3	The enemy ships will have a scripted A.I which shall scale corresponding to the player's score	**Mandatory**

3.3.1	The enemy ships will move the left and right edges of the screen at a fixed rate and will not move outside of the horizontal window boundary (0-400).	**Mandatory**
3.3.2	The enemy ships will have an increasing erratic movement behaviour as the player's score increases	**Desirable**
3.3.3	The enemy ships will have an increasing fire rate behaviour as the player's score increases	**Desirable**
3.5	The enemy will fire a primary weapon	**Mandatory**
3.5.1	The enemy will fire a secondary weapon	**Optional**
3.6	The enemy will be destroyed when a collision occurs between a projectile detection box and the enemy detection box	**Mandatory**
3.6.1	The enemy will be destroyed when a collision occurs between the player detection box and the enemy detection box	**Mandatory**
3.6.2	When an enemy dies, the player's score will be increased by a value of '1'	**Mandatory**

Power ups

4.0	A power up system will be implemented into the game	**Desirable**
4.1	When an enemy dies, there is a 1/20 chance for a power up to spawn	**Desirable**
4.1.1	Power ups will move downwards in the game window	**Desirable**
4.1.1.1	Power ups will despawn when off screen	**Mandatory**
4.1.2	When a player collides with a power up, enable power up and remove the power up entity	**Mandatory**
4.2	There will be a variety of power ups available to the player	**Optional**

Non-functional Specifications

The non-functional specifications are required in the game as the requirements indicate that many users would like interface elements and a scoring/lives system displayed on screen for the benefit of the user

User Interface

5.0	The game will have various menu systems	**Desirable**

5.1	The game will have a main menu	**Mandatory**
5.1.1	The game will start up in the main menu	**Mandatory**
5.1.2	The main menu will have various options the player can select	**Desirable**
5.1.2.1	The player will be able to select ‘start game’	**Mandatory**
5.1.2.2	The player will be able to select ‘options’	**Optional**
5.1.2.3	The player will be able to select ‘controls’	**Desirable**
5.1.2.4	The player will be able to select ‘exit game’	**Optional**
5.2	The game will have a pause menu	**Desirable**
5.2.1	The pause menu will have various options the player can select	**Desirable**
5.2.1.1	The player will be able to select ‘resume game’	**Desirable**
5.2.1.2	The player will be able to select ‘main menu’	**Desirable**
5.2.1.3	The player will be able to select ‘options’	**Optional**

Health System

6.0	The game will have a health system	**Mandatory**
6.1	The health will be an integer value between 0 and 100	**Mandatory**
6.2	The player’s health will start at 100	**Desirable**
6.3	Health will be reduced when an enemy ship/projectile collides with the player sprite	**Mandatory**
6.4	When a player’s health is 0, a life is lost	**Mandatory**
6.5	Player’s health will be displayed on the bottom left	**Mandatory**

Life System

7.0	The game will have a life system	**Mandatory**
7.1	Lives will be an integer value	**Mandatory**
7.2	A player will initially have 2 lives	**Mandatory**
7.3	A life is lost when a player’s health reaches 0	**Mandatory**

7.4	The game is reset when a players lives is equal to 0	**Mandatory**
7.5	Player's number of remaining lives will be displayed on the bottom right	**Mandatory**

Scoring System

8.0	The game will have a score system	**Desirable**
8.1	The score will be an integer	**Desirable**
8.2	The score will increase every time an enemy ship dies	**Desirable**
8.3	The score will increase every time as power up is used	**Desirable**

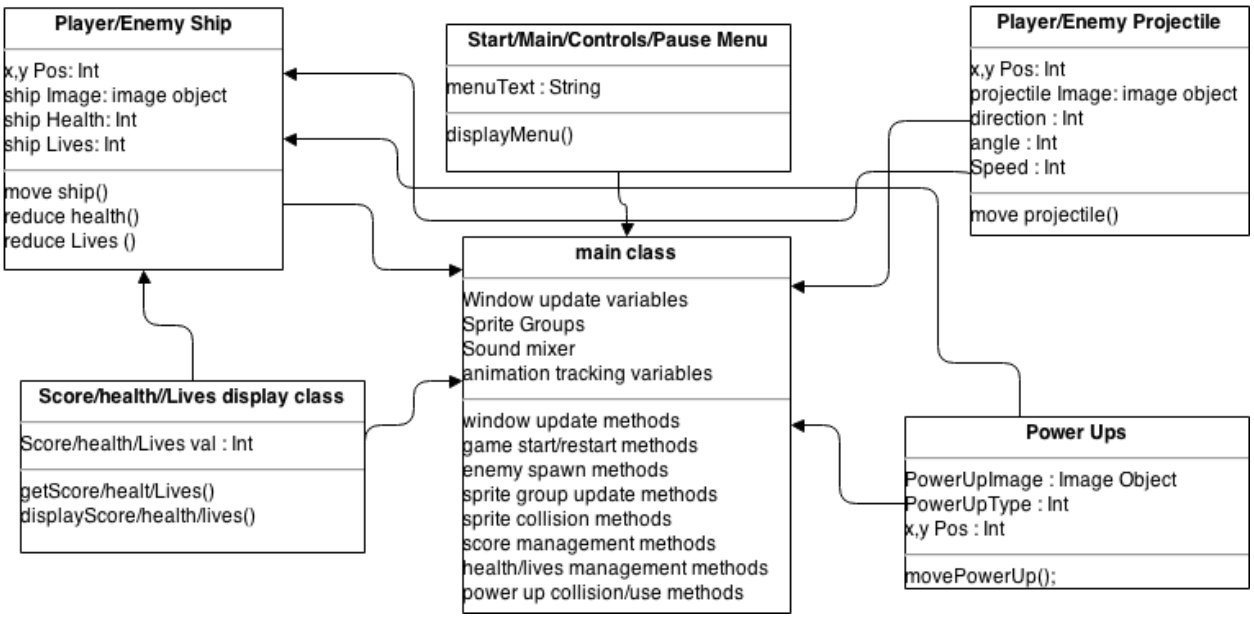
Game Design

Game Overview

'LOGIC WARS' implements many features of classic side-scrolling shoot 'em up games taking heavy influence from titles such as the successful arcade game '1942' from Capcom in 1984. The player controls a spaceship and fires upon enemies that descend towards the player, who are also returning fire. The primary aim is to earn a high score by destroying enemy ships and avoiding enemy fire. The player loses the game once all their lives have been depleted. A life is lost when the players health is reduced to 0 from enemy fire and/or collision with enemy ships. Various power ups have a chance to be spawned after enemy ships are destroyed to provide a temporary or lasting advantage. The game is stylised such that the ships are shaped like logic gates; and that the enemies are shaped like the negated equivalents (NOR logic gates) thus adding a satirical twist to the typical arcade game.

Interaction design

The game has a minimalistic design in its interaction between modules of the game. Each module has its own class and most classes are tied to the 'main' class whereby they are called and instantiated when the conditions are correct. Following is a UML depicting the interaction of the modules with each other.



This diagram shows the relations each of the types of classes have, all are used by the main class and the projectile/score/power up classes use the locations of the enemy/player ship classes in order to spawn in the correct place.

Controller Behaviour

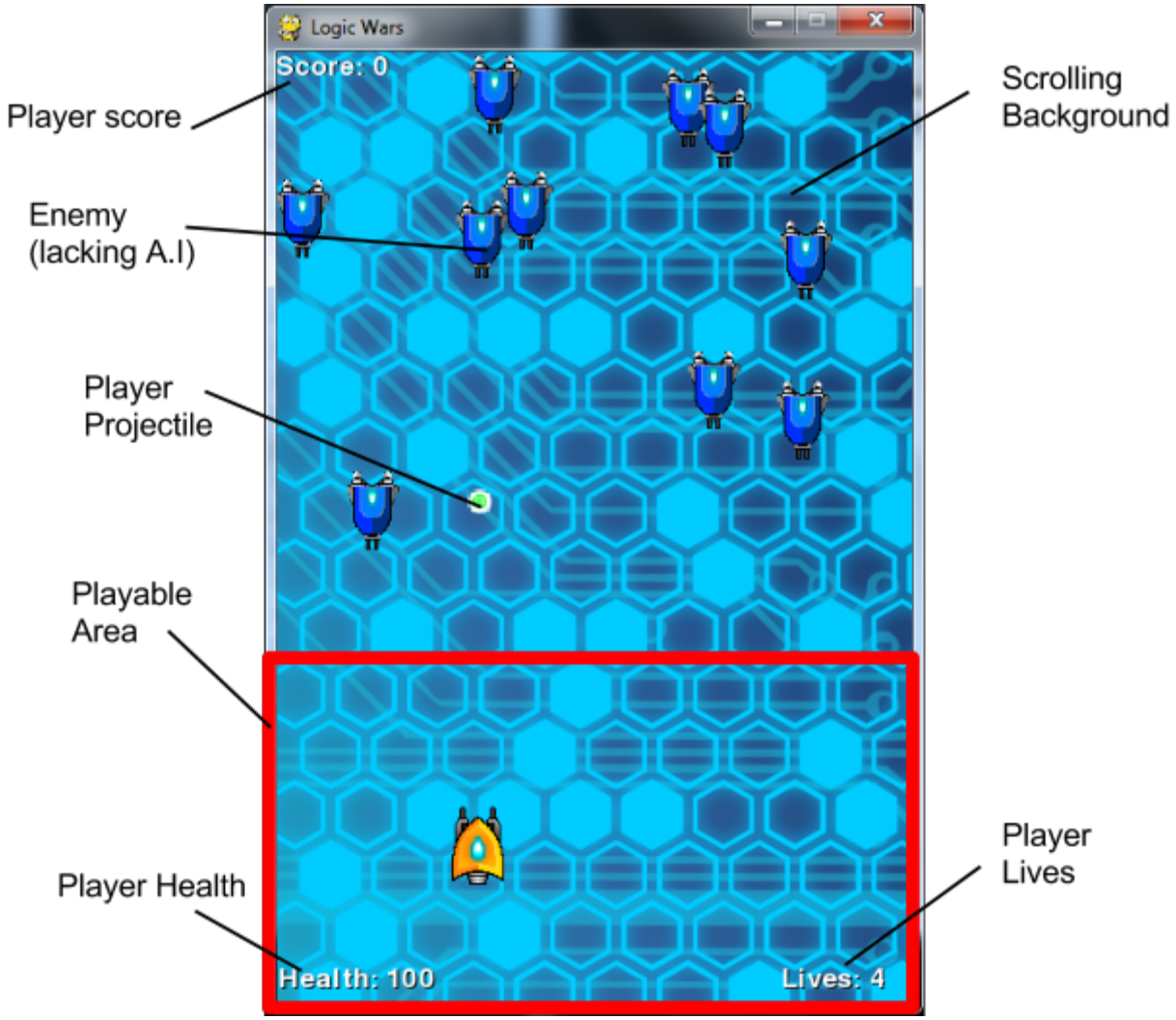
The game employes simplistic controls of any classical shoot em' up. you can control the player ship with the 'wasd' keys or the arrow keys on the keyboard and can press the spacebar to fire the primary weapon of your ship. You can pause the game with the 'p' key and access the controls menu from the main menu by pressing the 'c' key. On the main menu, you press the 'spacebar' to start and on the game over screen, you press the 'enter' key to restart the game.

Scoring Mechanism

The scoring mechanism is simple: if the player hits an enemy ship, the player score will increase by one if the player ship gets certain power ups, they will also obtain more points. This is purposely simple given that the hectic nature of the game will dictate that avoiding the projectiles will be a large focus of the game.

Evaluation of prototype and design refinements

Initially we had to present our progress to the client in a prototyping session. The game was expected to have its basic and/or main functionalities at a stable build with room for improvement and refining. In our prototype, we had the player representation almost completed, including the player region, player controls and fire mechanism.



Our prototype also includes the functionalities of score, health and lives yet could not be fully shown to the client due to lack of enemy A.I. The enemy spawning system worked but the enemies did not move nor fire back at the player.

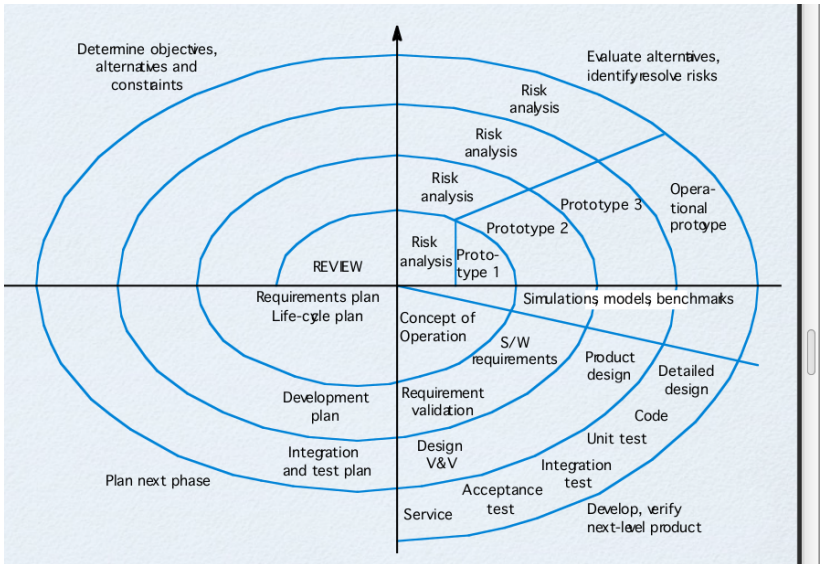
The feedback we gained focused mostly on animations. Our client told us we needed animation for both enemies and the player ship. Also, it was obvious that an A.I for the enemy was of utmost priority as they simply didn't move/ shoot or continually respawn. The general feel of the game was good but still had areas for development, especially in player interaction and response i.e menu's and animation.

Our design was refined such that new images would be included to ensure that the game actually had animation and motion as a key part of its functionality. the player, enemies and all projectiles would be given animations to ensure this.

Game Implementation

Development Methodology

In this project we would be adopting an agile development approach, using the **spiral model** with **extreme** and **pairs programming.** This methodology involves programming **in pairs** in a continuous process to code and refactor in order to develop a functional and minimalist program.

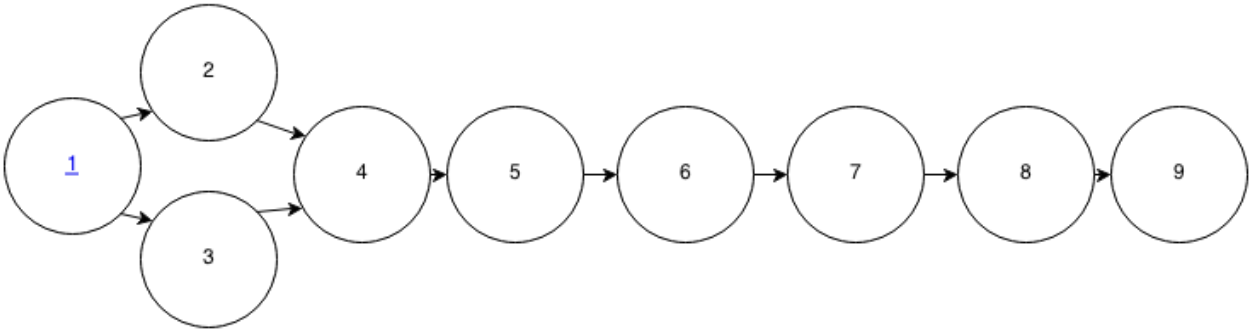


We aim to simulate 2-3 cycles of the spiral in order to fully realise and implement the requirements within the time/resource constraints we have. This involves the several review stages to ensure we are following and achieving the requirements in the sections above.

PERT Planning

The first course of action was to produce a Pert chart to organise our work load structure and to formulate our deadlines and targets effectively. The next page shows our pert chart, showing the estimations of the time taken for each task.

PERT Chart for the development progress



Pert chart number	Estimated time taken	Description
1	1 weeks	Come up with an original game concept; perhaps building from a traditional genre of game. Research games in the chosen genre that may or may not be developed using the pygame libraries in order to obtain an understanding of typica games and architectures in pygame
2	1 week	develop a high level model structure of the game - what elements and modules there will be and how they will interact
3	1 week	create a basic low-level design and implementation of the game we intend to create. This involves creating basic sprites and functions for the prototype
4	1 week	creation of the basic art assets for the game. This included sprite images for the player and the sprites for projectiles
5	1 day	prototype release. This was the initial release of the game as it was in its simplistic state. In order to obtain feedback from the user on the current state of the product and the requirement.
6	1 day	Refining requirements. With the feedback from the prototype demo day we refine the requirements.
7	1 weeks	using the refined requirements and the basic low level design create a detailed low-level design of the final game with all the features and functions implemented.
8	1 week	System testing: testing the product to ensure that it satisfies the requirements

9	1 week	do User acceptance testing to ensure that the customer approves of the final product; this is done with a user evaluation

After completing the requirements elicitation process and a plan of action, we proceeded to create a prototype product to refine the requirements and produce a final implementation.

Design implementation in code

The organisation of the games files is divided amongst two major folders known as Assets; which include our sound files and image files and our src; which includes all of our python source files containing various classes to cater for the needs of the game.

MainLoop.py

MainLoop.py file brings together all the code from the other classes contained in the other .py files. Without MainLoop, our game wouldn't function as it provides vital definitions for everything from the game itself to everything within the game. The MainLoop is what makes the game a game and not a long list of variables and blocks of code.
The game window requirements were relatively small and were easily implemented through our MainLoop.py file.

```
38      #create and define the screen window
39      self.screenHeight, self.screenWidth = 600, 400
40      self.window = pygame.display.set_mode([self.screenWidth, self.screenHeight])
41      pygame.display.set_caption("Logic Wars")
42      pygame.display.set_icon(pygame.image.load(
        "../Assets/Art/ORgateShipRevision1.png"))
43
```

Here, we declare the size of the window followed by the title which is accompanied by a small game logo.

The MainLoop file also creates the game state, game background and provides the generation of the sprites for both the player ship, enemy ships, scores, lives, healths and the menu systems. These are categorised into sprite groups and organised for ease of access.

```
#ship sprite
self.shipxPos, self.shipyPos = 200, 400
self.player = PlayerShip((self.shipxPos, self.shipyPos))
#health bar sprites
self.healthBarShadow = PlayerHealth("black", (51,586), self.player.health)

self.healthBar = PlayerHealth("white", (50,585), self.player.health)
self.playerScoreShadow = PlayerScore("black", (35,10))
self.playerScore = PlayerScore("white", (34,10))
self.playerLivesShadow = PlayerLives("black", (351,586), self.player.lives)

self.playerLives = PlayerLives("white", (350,585), self.player.lives)
#pause menu sprites
self.pauseText1 = PauseGameText("white", "PAUSED", 36, (200,200))
self.pauseText2 = PauseGameText("white", "press 'p' to return", 26, (200, 250
))
#main menu sprites
self.mainMenuText1 = PauseGameText("white", "Logic Wars", 36, (200, 200))
self.mainMenuText2 = PauseGameText("white", "Press SPACE to Start!", 26, (
200, 250))
self.mainMenuText3 = PauseGameText("white", "Press 'c' to see the controls",
26, (200, 300))
#control menu sprites
self.controlMenuText1 = PauseGameText("white", "SHIP CONTROLS", 36, (200,
200))
self.controlMenuText2 = PauseGameText("white", "w,a,s,d for ship movement",
 26, (200, 250))
self.controlMenuText3 = PauseGameText("white", "arrow keys for ship
movement", 26, (200, 300))
self.controlMenuText4 = PauseGameText("white", "SPACE to fire", 26, (200,
350))
self.controlMenuText5 = PauseGameText("white", "p to pause the game", 26, (
200, 400))
self.controlMenuText6 = PauseGameText("white", "press c to return to menu ",
 26, (200, 500))
self.controlMenuText7 = PauseGameText("white", "or SPACE to start game", 26
, (200, 520))
#game over sprites
self.gameOverText1 = PauseGameText("white", "GAME OVER", 36, (200, 200))
self.gameOverText2 = PauseGameText("white", "you lost the game", 26, (200,
400))
self.gameOverText3 = PauseGameText("white", "press ENTER to retry", 26, (
200, 420))
```

```
    self.oneUp = pygame.mixer.Sound("../Assets/SoundEffects/oneUp.ogg")

    #control mapping for player in game
    self.key_map = {
        pygame.K_w: [self.player.keyUp, self.player.keyDown],
        pygame.K_s: [self.player.keyDown, self.player.keyUp],
        pygame.K_a: [self.player.keyLeft, self.player.keyRight],
        pygame.K_d: [self.player.keyRight, self.player.keyLeft],
        pygame.K_UP: [self.player.keyUp, self.player.keyDown],
                pygame.K_DOWN: [self.player.keyDown, self.player.keyUp],
                pygame.K_LEFT: [self.player.keyLeft, self.player.keyRight],
                pygame.K_RIGHT: [self.player.keyRight, self.player.keyLeft]
    }

    #limit game clock
    self.gameClock = pygame.time.Clock()
    self.gameClock.tick(30)

    #spawns the first batch of enemies
    self.spawnEnemies()

    # draw and display on screen
    pygame.display.update()
    pygame.display.flip()

"""displays the main menu items"""
def mainMenu(self):
    self.spriteList.clear(self.window, self.screen)
    self.window.fill(pygame.Color("black"))
    self.mainMenuItems.update()
    self.mainMenuItems.draw(self.window)
    pygame.display.flip()

"""stars the game"""
def startGame(self):
    self.inMainMenu = not self.inMainMenu

"""displaus the control menu items"""
def controlMenu(self):
    self.spriteList.clear(self.window, self.screen)
    self.window.fill(pygame.Color("black"))
    self.controlMenuItems.update()
    self.controlMenuItems.draw(self.window)
    pygame.display.flip()

"""determines whether or not the player is in the control menu"""
def controlsEvent(self):
    self.inControlMenu = not self.inControlMenu

"""displays the pause menu items"""
def pauseMenu(self):
    self.spriteList.clear(self.window, self.screen)
    self.window.fill(pygame.Color("black"))
    self.pauseMenuItems.update()
    self.pauseMenuItems.draw(self.window)
    pygame.display.flip()
```

Furthermore, MainLoop loads our sound files, generates the game clock (speed) and control mapping system. It then defines how to update the game screen and defines the menus.

Finally, our MainLoop controls enemy spawns, unit collision and power up spawns - ensuring enemies despawn when hit and/or when moved off screen and power ups despawn when consumed or when moved off screen.

Player

PlayerShip

This file defines the specificities of the ships position, health, number of lives and representation. It defines the response to player controls and creates hit detection for the ship. The detection

```
        #define the ship image, health, lives and ship x/y coords
        self.shipImages = []
        self.xCoord = 0
        self.yCoord = 0                 #ship classification will change ship image
        self.health=100
        self.lives=2

        #load the image for the ship
        for i in range(1,5):
            self.curImage = pygame.image.load("../Assets/Art/ORgateShipRevision" + str(i) + ".png")
            self.curImage = pygame.transform.scale(self.curImage, (33, 49))
            self.shipImages.append(self.curImage)
            self.image = self.shipImages[0]

        #load the rectangle image behind the sprite
        self.rect = self.image.get_rect()
        self.rect.center = spawnCoords

    #method moves ship down
    def keyDown(self):
        self.yCoord += 3

    #method moves ship up
    def keyUp(self):
        self.yCoord -= 3

    #method moves ship left
    def keyLeft(self):
        self.xCoord -= 3

    #method moves ship right
    def keyRight(self):
        self.xCoord += 3

    #method reduces ship health upon hit                #need to implement varying health and enem
    def takeHit(self, damageVal):

        #if damage dealt is < current health: reduce current health
        if damageVal < self.health:
            self.health -= damageVal
        #else reduce lives and reset health counter
        else:
            self.lives -= 1
            self.health = 100

    def addHealth(self, newHealth):
        self.health += newHealth

    def addLives(self, newLives):
        self.lives += newLives

    #updates sprite state on screen - x/y coordinates
    def update(self):
        self.rect.move_ip(self.xCoord, self.yCoord)

        #animate the sprite with correct image
        if self.xCoord < 0:
            self.image = self.shipImages[2]
            self.image = pygame.transform.scale(self.image, (33, 49))
        elif self.xCoord > 0:
            self.image = self.shipImages[1]
            self.image = pygame.transform.scale(self.image, (33, 49))
        elif self.yCoord < 0:
            self.image = self.shipImages[3]
            self.image = pygame.transform.scale(self.image, (33, 64))
        else:
            self.image = self.shipImages[0]
            self.image = pygame.transform.scale(self.image, (33, 49))

        #define the boundaries in which the ship can move
        self.rect.top = max(350, self.rect.top)
        self.rect.bottom = min(575, self.rect.bottom)
        self.rect.left = min(367, self.rect.left)
        self.rect.right = max(33, self.rect.right)

```

also corresponds to damage values to affect health and lives. This file also ensures that the x and y coordinates of the ship are updated.

In correspondence to movement through player input, the file also updates the image representation of the ship dependent on the direction, referencing image files.

Finally, this file also defines the playable zone for the player ship; which is a necessary requirement for the game.

Projectile

The file enlists the details for a projectile entity which is spawned from the player ship location. It also creates a detection box for the projectile to ensure hit detection functions properly between itself and enemy ships.

```python
#import game modules
import sys, pygame
from pygame.sprite import Sprite
from pygame.locals import *

"""class that defines and characterises projectile objects within the game"""
class Projectile(Sprite):

#need to do different types of object and strength


    #initialises projectile object upon invocation
    def __init__(self, spawnCoords):
            pygame.sprite.Sprite.__init__(self)

        self.counter = 0
        self.projectileClock = 0
        self.bulletImages = []
            #load the image for the ship
            for i in range(1,3):
            self.curImage = pygame.image.load("../Assets/Art/plasmaProjectile" + str(i)  + ".png")
                self.curImage = pygame.transform.scale(self.curImage, (15, 15))
            self.bulletImages.append(self.curImage)
            self.image = self.bulletImages[0]

        #load the rectangle image behind the sprite
            self.rect = self.image.get_rect()
        self.rect.center = spawnCoords

    #method updates sprite state - moves projectiles along a linear path
    def update(self):
        self.rect.y -= 5

        if self.projectileClock == 0  and self.counter == 40:
            self.image = self.bulletImages[0]
            self.counter = 0
            self.projectileClock = 1
        elif self.projectileClock == 1 and self.counter == 40:
            self.image = self.bulletImages[1]
            self.counter = 0
            self.projectileClock = 0
        else:
            self.counter += 1
```

PlayerScore, PlayerLives and PlayerHealth

These files define the PlayerScore, PlayerLives and PlayerHealth classes respectively which creates the specificities for the render location for the respective location, font colour and size. It also defines how to increase/decrease these values and updates them in real time.

```python
#import game modules
import sys, pygame
from pygame.sprite import Sprite
from pygame.locals import *

"""class to measure and display player score on screen"""
class PlayerScore(Sprite):

    #initialises the object upon invocation
    def __init__(self, colour, spawnCoords):
        pygame.sprite.Sprite.__init__(self)

        #set the initial score and font of the score
        self.colour = pygame.Color(colour)
        self.score = 0

        #sets the rect and render coords for score sprite
        self.font = pygame.font.Font(None, 26)
        self.renderText()
        self.rect = self.image.get_rect()
        self.rect.center = spawnCoords

    #simply renders the text on screen
    def renderText(self):
        self.image = self.font.render("Score: %d" % self.score, True, self.colour)

    #method increases player score                    #get multiplier sorted
    def increase(self):
        self.score += 1

    def increase2(self, newScore):
        self.score += newScore

    #method updates sprite state on screen
    def update(self):
        self.renderText()
```

```python
#import game modules
import sys, pygame
from pygame.sprite import Sprite
from pygame.locals import *

"""class to display and manage the state of the player lives"""
class PlayerLives(Sprite):

    #default constructor to define the state of the lives of the sprite
    def __init__(self, colour, spawnCoords, playerLives):
            pygame.sprite.Sprite.__init__(self)

        #define the colour and lives of the text display
            self.colour = pygame.Color(colour)
            self.lives = playerLives

        self.font = pygame.font.Font(None, 26)
            self.renderText()
            self.rect = self.image.get_rect()
            self.rect.center = spawnCoords

    #method that renders the text on screen
    def renderText(self):
            self.image = self.font.render("Lives: %d" % self.lives, True, self.colour)

    #method to decrease the player lives
    def decrease(self):
            self.lives -= 1
            self.renderText()

    #method to increase player lives
    def increase(self):
        self.lives += 1
        self.renderText()

    def newLives(self, newLives):
        self.lives = newLives

    #method to update the state of the object on screen
    def update(self):
        self.renderText()
```

Enemies

```python
#import game modules
import sys, pygame
from pygame.sprite import Sprite
from pygame.locals import *
"""method defines the state and behaviour of various enemy ships"""
class EnemyShip(Sprite):

#need different classes of ships and enemy mothership

    #default constructor - initisalises ship object on creation
    def __init__(self, spawnCoords, enemyShipType):
        pygame.sprite.Sprite.__init__(self)

        #set the initial image state and ship x/y Coords
        self.shipImages = []
        self.xCoord = 0
        self.yCoord = 0
        self.moveCounter = 0
        self.moveDirection = 0 #0 is left, 1 is right
        self.shipType = enemyShipType #determines the way in which the ship moves downscreen
        #load the image for the ship
        #image, ship health will change with classification

        #load the image for the ship
            for i in range(1,5):
                self.curImage = pygame.image.load("../Assets/Art/enemyNORGateShip" + str(i) + ".png")
                self.curImage = pygame.transform.scale(self.curImage, (33, 49))
          self.curImage = pygame.transform.rotate(self.curImage, 180)
                self.shipImages.append(self.curImage)
                self.image = self.shipImages[0]


        #get the sprite rect and render coords
        self.rect = self.image.get_rect()
        self.rect.center = spawnCoords

    #method moves the ship right
    def moveRight(self):
        self.xCoord += 1

    #method moves the ship left
    def moveLeft(self):
        self.xCoord -= 1

    #method moves the ship down
    def moveDown(self):
        self.yCoord += 1

    #method moves the ship up
    def moveUp(self):
        self.yCoord -= 1

    def moveSprite(self):
        self.moveCounter += 1

        if self.shipType == 0:
            if self.rect.left == 0:
                self.moveDirection = 1
                self.moveRight()
                self.rect.y += 10
                self.moveCounter = 0
            elif self.rect.left == 367:
                self.moveDirection = 0
                self.moveLeft()
```

EnemyShip

This file defines the EnemyShip sprite which renders the ship's location and image. It also specifies how the ship will move based on conditions and location.

EnemyProjectile

The file enlists the details for a projectile entity which is spawned from the enemy ship location. It also creates a detection box for the projectile to ensure hit detection functions properly between itself and the player ship.

```python
#import game modules
import sys, pygame
from pygame.sprite import Sprite
from pygame.locals import *

"""class that defines and characterises projectile objects within the game"""

class EnemyProjectile(Sprite):

    #initialises projectile object upon invocation
    def __init__(self, spawnCoords):
            pygame.sprite.Sprite.__init__(self)

        self.counter = 0
        self.projectileClock = 0
        self.bulletImage = None

            #load the image for the ship
        self.bulletImage = pygame.image.load("../Assets/Art/enemyPlasmaProjectile.png")
            self.bulletImage = pygame.transform.scale(self.bulletImage, (15, 15))
        self.image = self.bulletImage

        #load the rectangle image behind the sprite
            self.rect = self.image.get_rect()
        self.rect.center = spawnCoords

        #method updates sprite state - moves projectiles along a linear path
    def update(self):
        self.rect.y += 2

        if self.projectileClock == 0  and self.counter == 40:
            self.counter = 0
            self.projectileClock = 1
        elif self.projectileClock == 1 and self.counter == 40:
            self.counter = 0
            self.projectileClock = 0
        else:
            self.counter += 1
```

Power Ups

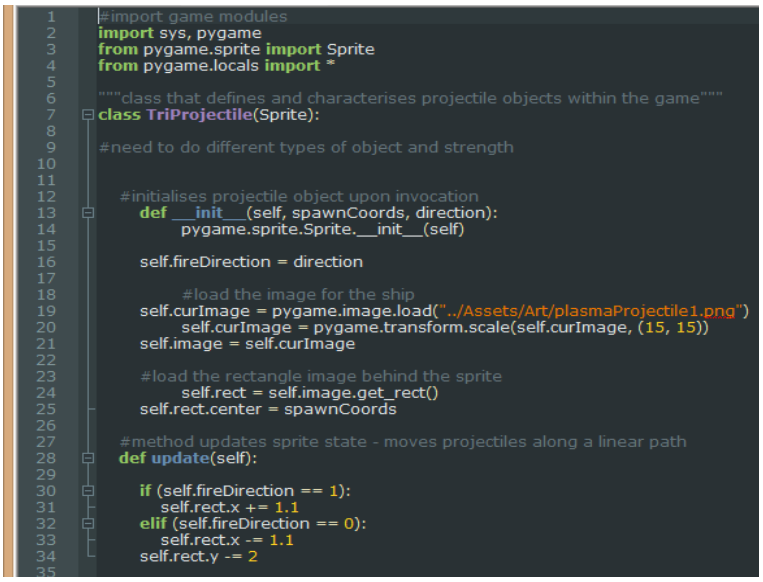
LevelUpToken

This file declares the PowerUp class which increases the player's number of lives by one.

```python
#import game modules
import sys, pygame
from pygame.sprite import Sprite
from pygame.locals import *

"""class that defines and a level up token that increases player Live"""

class PowerUp(Sprite):

    #initialises Token object upon invocation
    def __init__(self, spawnCoords, powerUpType):
            pygame.sprite.Sprite.__init__(self)

        #tokenNumber used to determine which type of token is used
        self.tokenType = powerUpType
        self.bulletImage = None

            #load the image for the Token

        #need to create image
        self.bulletImage = pygame.image.load("../Assets/Art/PowerUp" + str(powerUpType) + ".png")
            self.bulletImage = pygame.transform.scale(self.bulletImage, (20, 20))
        self.image = self.bulletImage

        #load the rectangle image behind the sprite
            self.rect = self.image.get_rect()
        self.rect.center = spawnCoords

    def update(self):
        self.rect.y += 1
```

TriProjectile

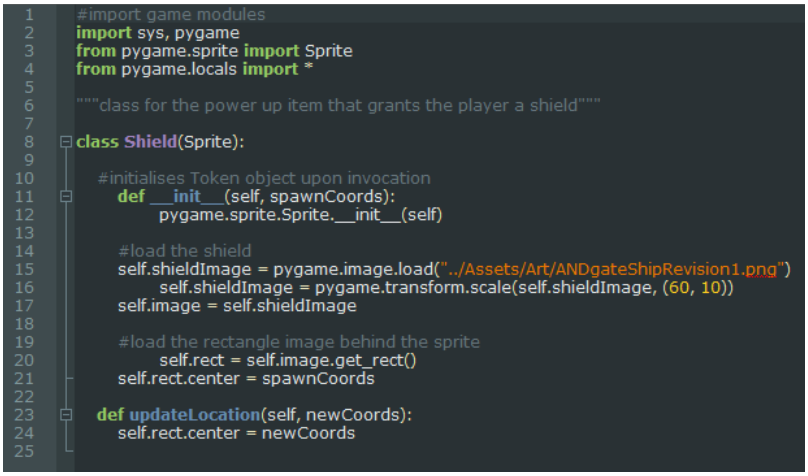
This file declares the TriProjectile class which grants the player the ability to fire three projectiles as opposed to just one. This occurs once per power up the player has picked up.



```
#import game modules
import sys, pygame
from pygame.sprite import Sprite
from pygame.locals import *

"""class that defines and characterises projectile objects within the game"""
class TriProjectile(Sprite):

#need to do different types of object and strength


    #initialises projectile object upon invocation
        def __init__(self, spawnCoords, direction):
            pygame.sprite.Sprite.__init__(self)

        self.fireDirection = direction

            #load the image for the ship
        self.curImage = pygame.image.load("../Assets/Art/plasmaProjectile1.png")
            self.curImage = pygame.transform.scale(self.curImage, (15, 15))
        self.image = self.curImage

        #load the rectangle image behind the sprite
            self.rect = self.image.get_rect()
        self.rect.center = spawnCoords

    #method updates sprite state - moves projectiles along a linear path
    def update(self):

        if (self.fireDirection == 1):
            self.rect.x += 1.1
        elif (self.fireDirection == 0):
            self.rect.x -= 1.1
        self.rect.y -= 2
```

Shield

This file declares the Shield class which grants the player a shield and loads the image of the shield on the ship. The shield blocks one incoming enemy projectile and then dissipates.

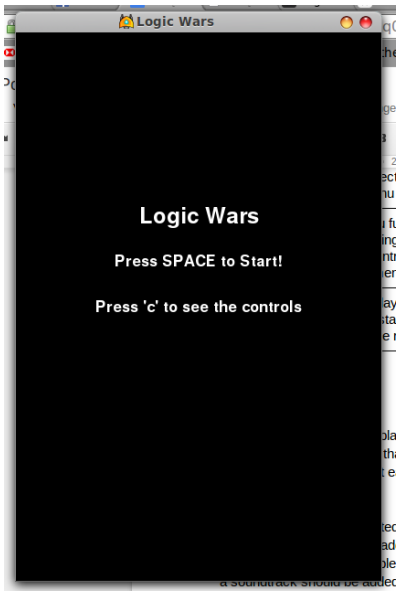


```
#import game modules
import sys, pygame
from pygame.sprite import Sprite
from pygame.locals import *

"""class for the power up item that grants the player a shield"""

class Shield(Sprite):

    #initialises Token object upon invocation
        def __init__(self, spawnCoords):
            pygame.sprite.Sprite.__init__(self)

        #load the shield
        self.shieldImage = pygame.image.load("../Assets/Art/ANDgateShipRevision1.png")
            self.shieldImage = pygame.transform.scale(self.shieldImage, (60, 10))
        self.image = self.shieldImage

        #load the rectangle image behind the sprite
            self.rect = self.image.get_rect()
        self.rect.center = spawnCoords

    def updateLocation(self, newCoords):
        self.rect.center = newCoords
```

Testing and Debugging

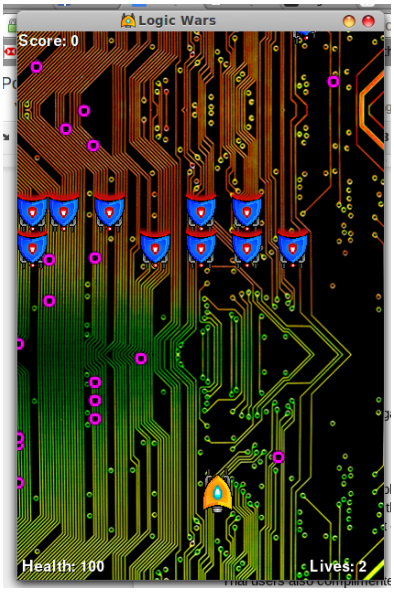
Test Number	Purpose	Pass-mark	Comments	Actions Required
1	Game works in functional window with correct resolution and title bar display	Pass	Consider an icon for the game	No immediate action necessary, icon is provided from player sprite
2	The player can navigate the ship with keyboard inputs and fire projectiles	Pass	Works with both arrow keys and w,a,s,d. Projectiles fire correctly.	None
3	The game has enemy ships which spawn, shoot and move towards the player	Pass	Spawns based on certain conditions, enemy projectiles could be faster	For now, the sheer number of projectiles caters for the slow speed
4	The game increases in difficulty based on player score	Fail	A.I is based on conditions of already spawned enemies	Alternative route taken
5	The player correctly loses lives until 0 when health is reduced to 0	Pass	Correctly allows a final life when displayed ‘0’ lives (remaining).	Perhaps some game representation of a death
6	The players health is reduced when hit by enemy ship/projectile	Pass	Correctly loses health	None
7	The players score increases with each enemy kill	Pass	-	None
8	Sprite entities are correctly deleted upon death/contact	Pass	There is no lag when large quantity of enemies are dying	None
9	Power ups spawn randomly on enemy deaths	Pass	Correctly spawns and is consumed	None

10	Power up: Shield correctly protects against one enemy projectile correctly	Pass	Correctly spawns and is consumed	None
11	Power up: Level Up correctly adds one extra life to the player	Pass	Correctly spawns and is consumed	None
12	Power up: TriProjectile correctly fires a triple projectile when used	Pass	Correctly spawns and is consumed	None
13	Power up: Health correctly adds health to the current players health	Pass	Correctly spawns and is consumed	None
14	The game correctly starts up in the main menu mode	Pass	Blank screen with instructions	None
15	The main menu functions correctly, allowing navigation between the control screen and the main menu	Pass	-	None
16	The game displays game over and offers a restart once the players lives are reduced	Pass	Works perfectly	Score could be moved to the center

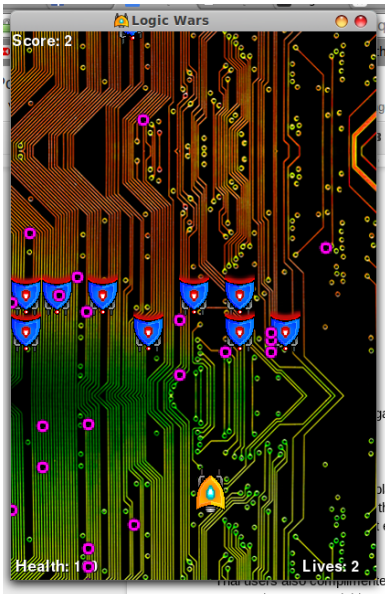


Evidence for tests 1 and 14: Window is correct resolution and icon added and the main menu displayed when ‘LogicWars.py’ is run.

Evidence for test 2 and 15: the game control menu shows all controls programmed into the game and is accessed via the main menu. The images below are testament to the functionality of controlling the ship.



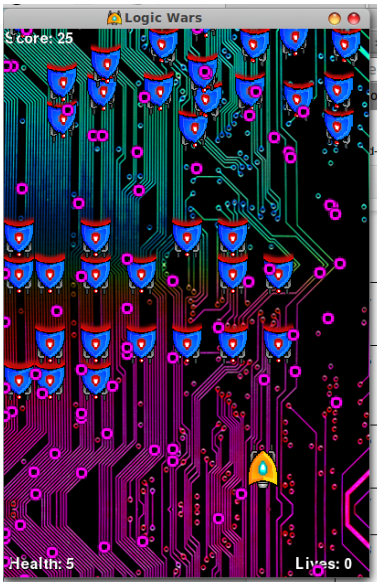
Evidence for test 2: before firing the projectile



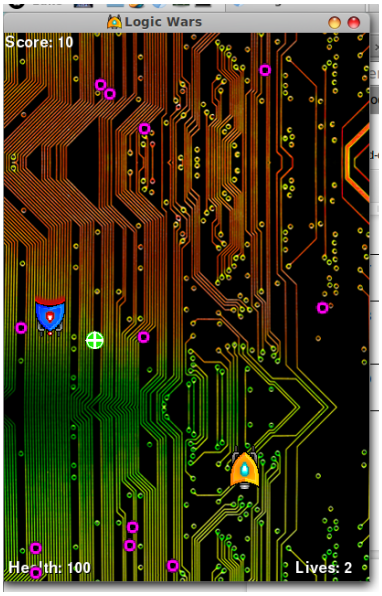
Evidence for test 3: after firing and hitting an enemy, enemy despawns (evidence of test 8) and the score increases (evidence for test 7). Compared to the previous test image, the enemy ships have moved whilst adopting the ‘group movement’ pattern. and more bullets have been fired by the ships.

```
#if you lose the game - display game over screen
elif self.gameOver:
        self.gameOverMenu()
        self.readGameOverControls()

#whilst the game is not paused or in the main menu or on the game over screen
if (not self.pausedGame and not self.inMainMenu and not self.gameOver):

        #read controls for the player ship
        self.readGameControls()

        #if the player runs out of lives - game over
        if self.player.lives == -1:
                self.gameOver = True
                self.gameOverSound.play()

        #if there are <=5 enemies left, spawn more
        if (self.enemiesRemaining <= 5):
                self.spawnEnemies()

        #adjust the current coords of a shield, if the player has it
        if self.hasShield:
                self.shield.updateLocation((self.player.rect.x + 15, self.player.rect.y - 2))

        #move the background image - yTranslation
        if self.changeFrame:
                self.yScaler += 1
                self.changeFrame = False#moves every other frame
        else:
                self.changeFrame = True

        if self.yScaler == 1400:
                self.yScaler = 0
```

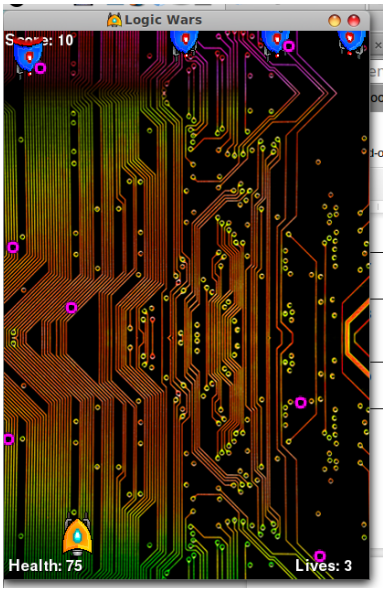
Evidence for test 4: (reference code) when there is less than or equal to 5 enemies left, more will spawn using the spawnEnemies() method in the mainLoop class



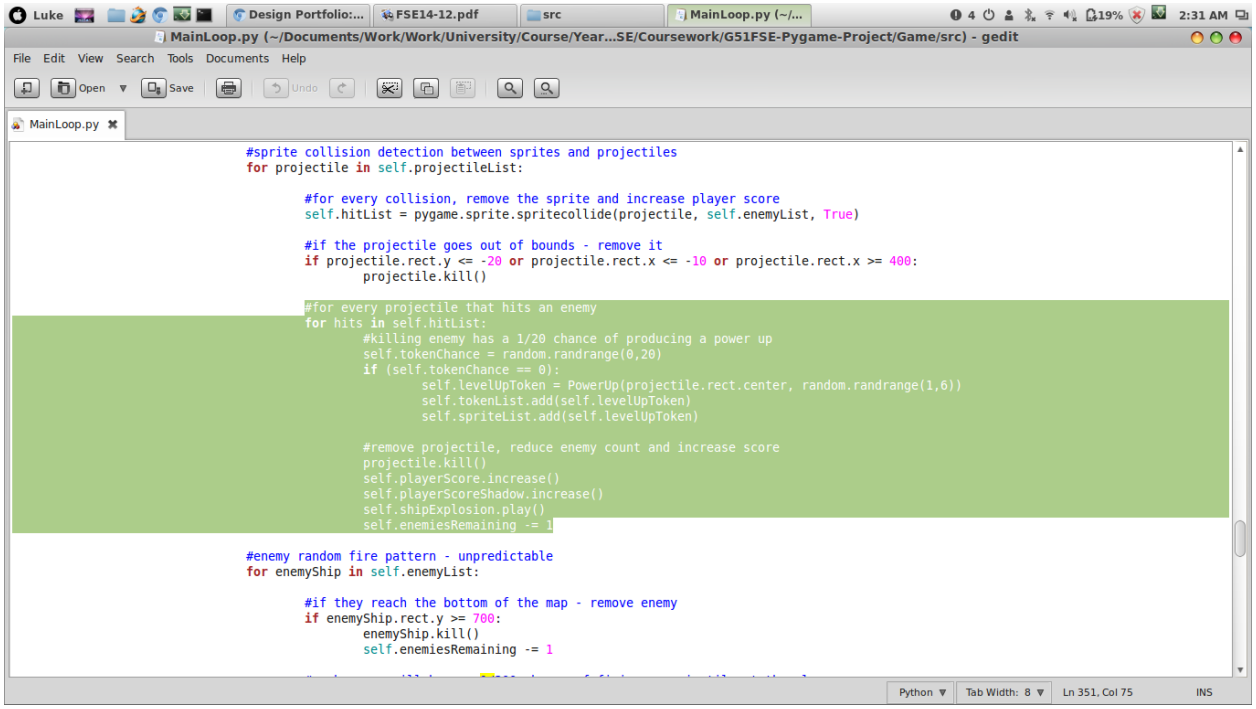
Evidence of test 5: the players ship health has been reduced to 0 lives and 5 health. Evidence for test 6: compare with previous images to see health has reduced as the player is hit by the enemy projectiles (purple).



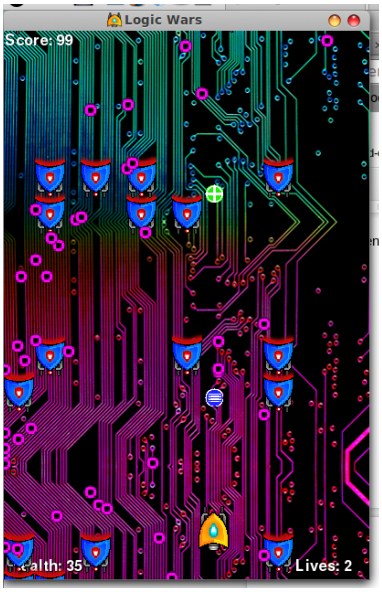
Evidence of tests 9 and 11: after firing upon some enemies, the ‘add life’ power up appears and scrolls down the screen. This power up gives the player another life.



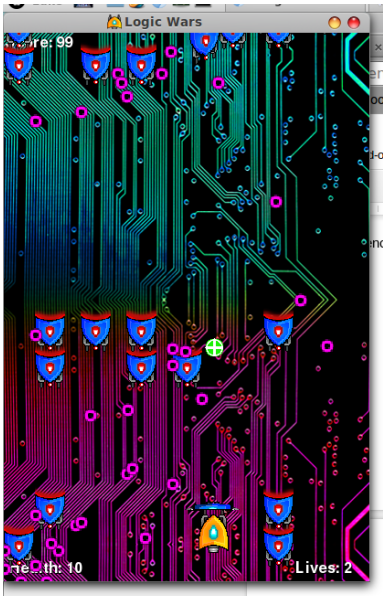
Evidence of test 9: After taking some damage from colliding with the enemy ship, the player collects the power up and hence the player lives (bottom left) has increased by 1.



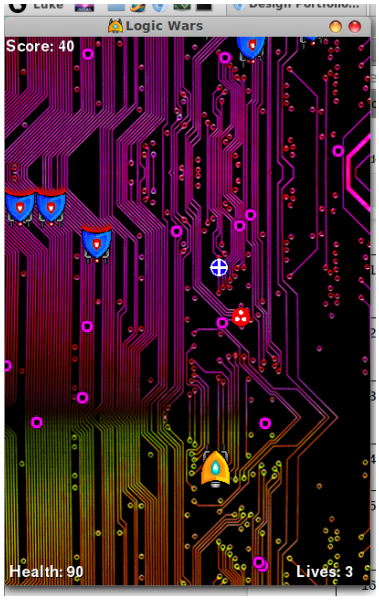
Evidence of test 9: The highlighted code shows that there is a 1/20 chance of producing a power up with each hit of an enemy ship. The python.random class is used to randomise the chance of obtaining said power up. There is also a random number from 1 to 6 generated which determines the type of power up created - each assigned a special rule (power up, shield etc…).



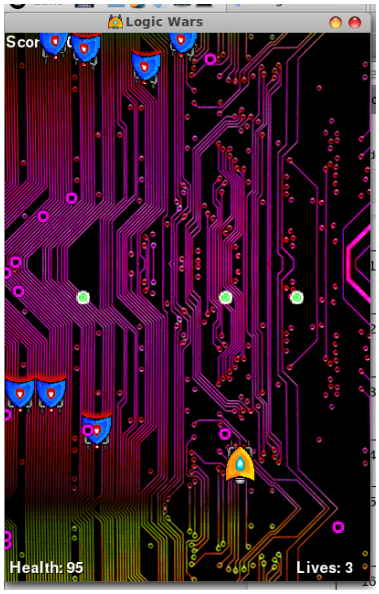
Evidence for test 10: the player successfully fires and kills an enemy, the shield power up spawns (blue circular power up present above player on screen) and is scrolling down screen towards the player.



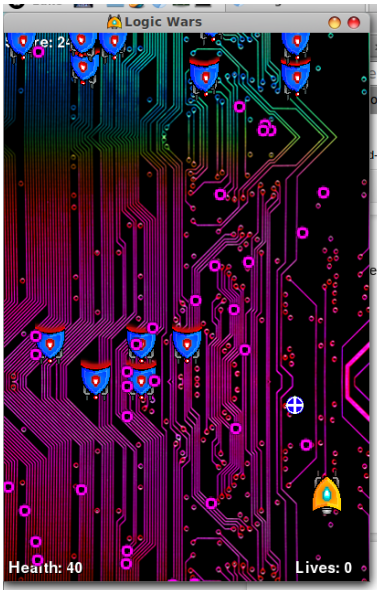
Evidence for test 10: after the player has obtained the shield, it is used and the shield appears above the player.



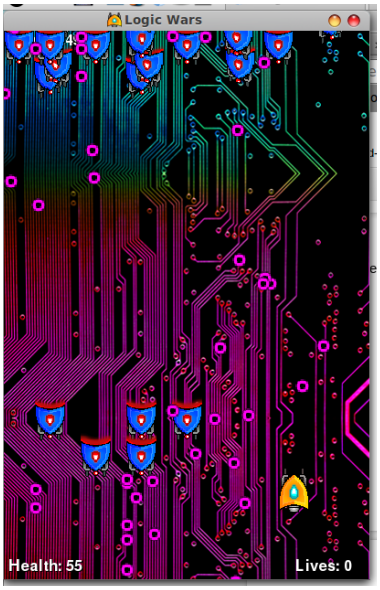
Evidence of test 12: the player shoots an enemy ship which despawns and produces a tri projectile power up (the red circular object above and approaching the player).



Evidence of test 12: the player collects the power up and fires 3 simultaneous projectiles as a result.



Evidence of test 13: The player hits an enemy ship which despawns and produced a health power up (blue circular object above player ship).



Evidence of test 13: The player collects the power up and the health increases from 40 to 55.

Evidence of test 16: the player loses all health and the game over screen is displayed. The player is given the choice to retry or can simply close the game to finish.

User Evaluation

From users, we found that players especially enjoyed the art style, animations and colourful background. Players found that the enemies were a challenge in large quantities and the game got hectic quickly, making it easy to dive into the game, accomplishing what user stories research suggested.

Trial users also complimented the response and quickness of the ship and its ability to fly across the screen quickly, adding to the general ‘epic’ feel of the game. Players also appreciated the fixed playable zone. To accommodate this ‘epic’ feeling, the user exclaimed that a soundtrack should be added however this would be too time consuming and inefficient for the games development as a loyalty free soundtrack would have to be found or a soundtrack would have to be composed by the team.

A few users suggested that the A.I be improved, most saying that there was a consistent pattern that could easily be exploited or defeated once one is attuned to the game. They said it didn’t remove any enjoyment from the game, but would add more if the A.I was much more randomised.

One player made the comment that the game should increase in difficulty. Reflective of our requirements, this seems the next step in improving the game as it would be eliminate the current ‘pattern’ A.I and also increase difficulty in another dimension. Furthermore, the addition of varied enemies seemed popular and would be another area to improve on to further enhance the game.

References

Here is a list of the content used in the game for assets.

- ship fire 1, by ani_music | License: Attribution
http://www.freesound.org/people/ani_music/sounds/219620/
- ship explosion, by wubitog | License: Creative Commons 0
http://www.freesound.org/people/wubitog/sounds/200465/
- bomb, by tlwm | License: Attribution
http://www.freesound.org/people/Dpoggioli/sounds/196914/
- triProjectile, by Dpoggioli | License: Attribution
http://www.freesound.org/people/tlwm/sounds/165825/
- shield, by peepholecircus | License: Creative Commons 0
http://www.freesound.org/people/peepholecircus/sounds/171705/
- one up, by D W | License: Attribution
http://www.freesound.org/people/D%20W/sounds/143607/
- add health by jobro | License: Attribution
http://www.freesound.org/people/jobro/sounds/33789/
- game Over, by jivatma07 | License: Creative Commons 0
http://www.freesound.org/people/jivatma07/sounds/173859/