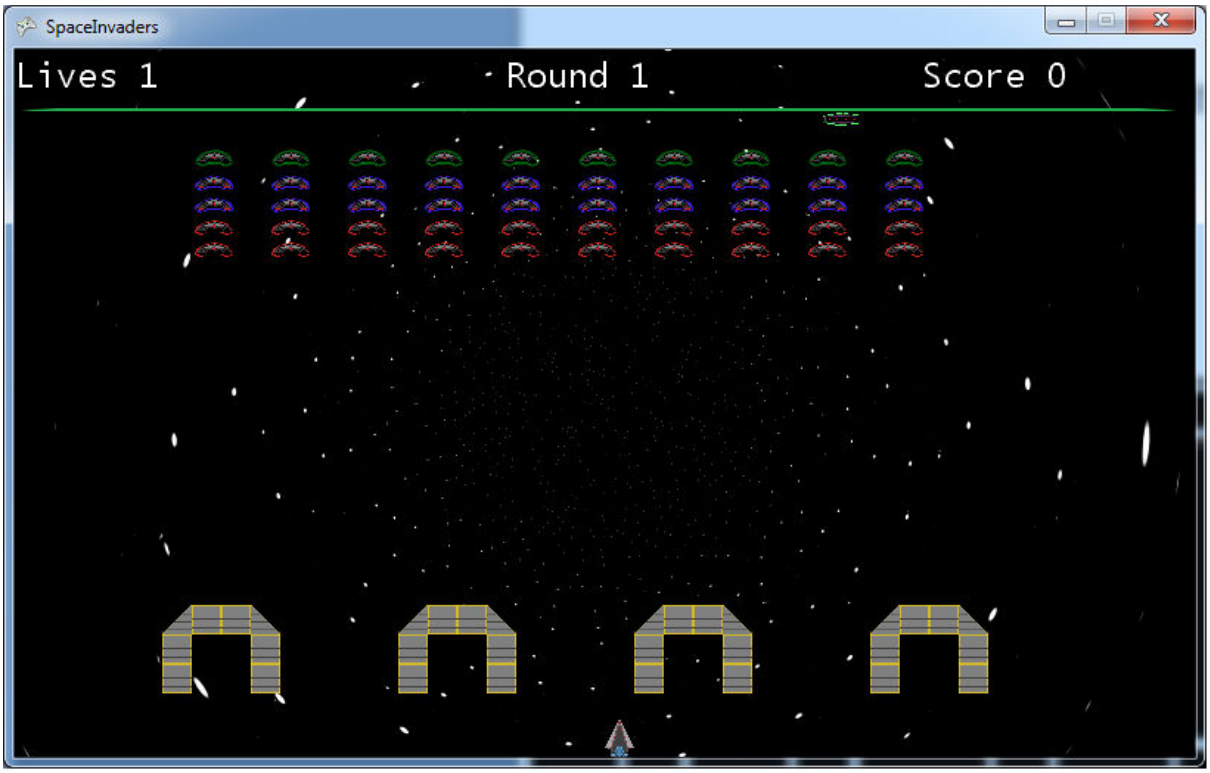
Final Report

2 Dimensional, Multiplatform Space Invaders Game Using Microsoft XNA Game Studio



Written by Stephen McQueen, Supervised by Dr F.C. Langbein and moderated by Prof R.R. Martin

Cardiff University School of Computer Science and Informatics

Date of Completion: 4th May 2012

Abstract

The aim of this project is to take a fresh look at a classic arcade video game and attempt to re-implement it using a modern programming environment along with modern programming techniques and devices to be compatible with multiple hardware platforms.

This project report will take you through the logical steps undertaken to appreciate and understand the game as well as to illustrate the challenges faced when implementing my own version of the game. Ultimately this project will attempt to "improve" on the original game by increasing its compatibility with hardware platforms, and by building on the games original functionality to create an overall more enjoyable gaming experience.

Acknowledgements

This project would not be feasibly achievable without the continuous help and support and feedback provided to me by my project supervisor Dr.F. Langbein. Dr. Langbein has provided me with invaluable advice in almost all aspects of this project as well as motivating me to push this project to its absolute limit whilst providing much encouragement and enthusiasm throughout.

Finally I would also like to thank Microsoft and their website create.msdn.com/en-US/ along with its community for providing me with a free programming environment (Visual Studio 2011) and frameworks (.NET, XNA), along with highly detailed tutorials and samples of which have proven to be absolutely crucial to the overall success and feasibility of the project.

Contents

Introduction

The first aim of this project is to implement a clone version of the original Space Invaders arcade game using a modern programming environment and modern programming techniques which is compatible with numerous hardware platforms.

The second aim of the project is to then build upon the original game and attempt to improve it by making noticeable game play changes that ultimately allows for a more enjoyable and less predictable gaming experience.

The interim report includes an in depth research of all aspects of the original Space Invaders, covering its history, challenges and potential areas for improvement. It also covers an in depth study with regards to implementing some form of "game A.I" to the modern version of the game and how this could be achieved.

This report will cover in depth the sheer scale of the project and dissect it into small and concise sections. It will cover the challenges faced during the implementation stage and provide an insight to the program source code as well as in game screenshots to illustrate the project progress and solution. Finally the report will reflect on the learning outcomes gained and whether the program meets the specification outlined in the Interim Report.

Design

Deliverables Recap

Before tackling any design of the game it is crucial to note the main objectives (deliverables) that the solution must satisfy in the form of a specification. Based on my previous research my high level specification for the program is the following:

- Be multi platform; namely for the Xbox 360, Windows OS PC and Windows OS Phone.
- Include a playable clone of the original game.
- Include a modernistic version of the game.

Entity Relationship Diagram

The first step in the design phase is to identify any potential objects that would need to be implemented in my version of the game. This is easily achievable by playing the original game a few times over and the notable objects are:

- Player
- Enemies
- Mystery Enemy
- Laser
- Enemy Laser
- Barriers
- Menu screen, high score screen, play screen.

The next step is to then map the relationships between the potential objects to gain an understanding of how these objects would interact with one another. In order to do this an

entity relationship diagram has been created (otherwise known as ERD) to map the relationships and visually express them.

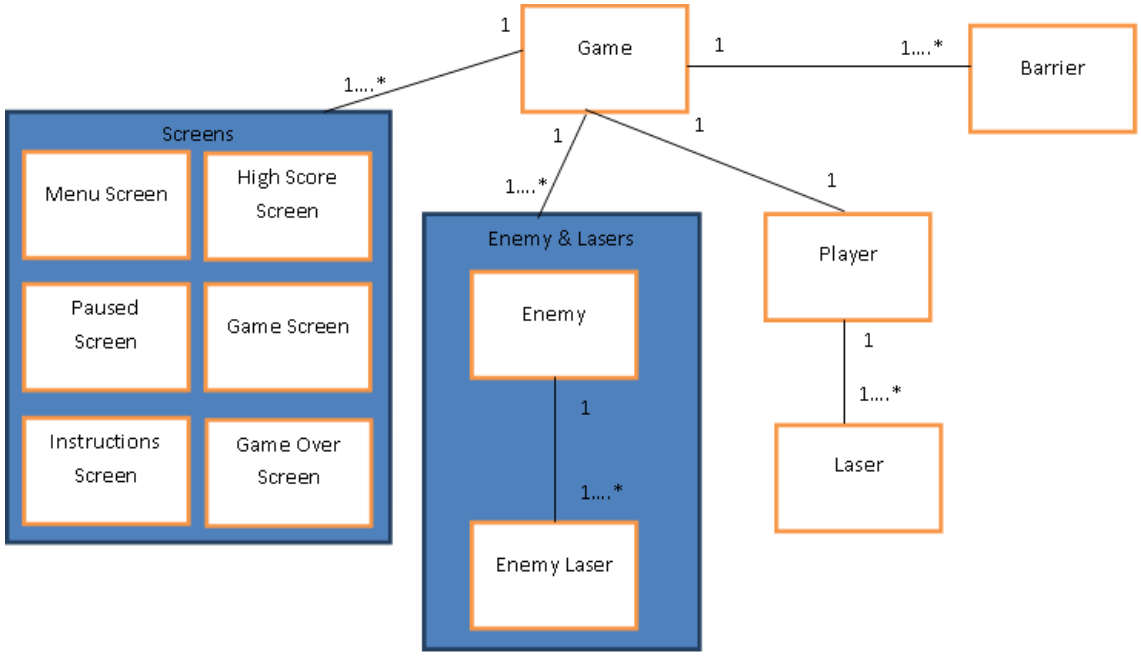


Figure 1 – Entity Relationship Diagram

You may notice in the ERD that a game object has been added which acts as a central hub to all objects involved. As mentioned in the interim report, in most games (if not all) there exists a central "game loop" object in which controls the continuous running of the game in any state except when the player opts to exit the game entirely. Having this central "Game" object normalises the game data such that there is always a singular relationship between any object and the Game object whereas oppositely the Game object can have any relationship to any object. This therefore compliments the suggested game engine architecture.

Game Architecture

The next stage is to plan where and how these elements will fit into the game engine architecture provided by XNA Game Studio. The provided architecture is shown in the form of a data flow diagram below:

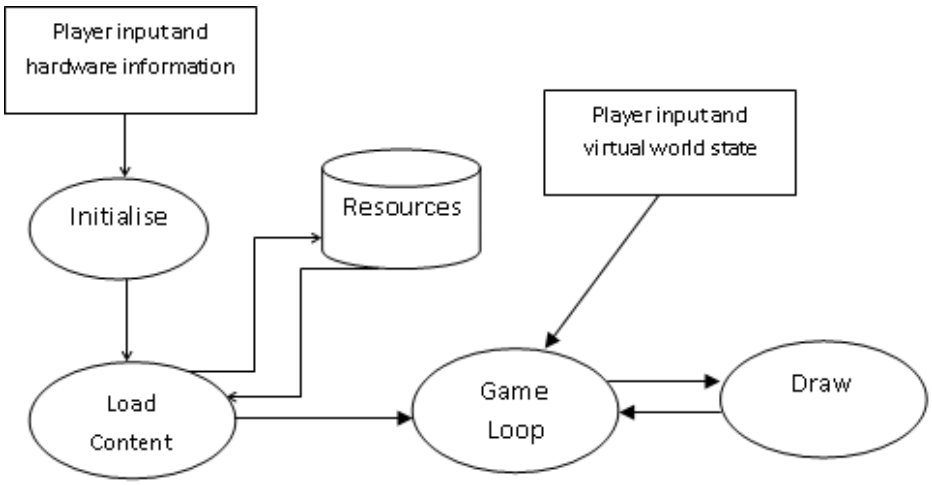


Figure 2 – High Level Data Flow Diagram

From Figure 2, the game engine can be broken down into three basic steps:

- Initialise
- Load Content
- Game Loop

Initialise

The initialise part of the architecture is used to declare variables and instantiate classes. In Layman's terms this stage is used to "set up" the game before running it. A data flow diagram of this part of the engine is shown below:

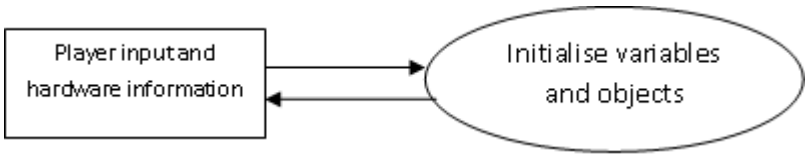


Figure 3 – Low Level Data Flow Diagram

As you can see from Figure 3, this part of the engine simply takes hardware specification information from the device and loads the game in readiness to be played.

Load Content

The load content part of the engine is very similar to the initialise stage but with one major difference. This part of the engine is solely dedicated to loading any graphical textures/fonts/animations from the internal storage device ready to be used by the game.

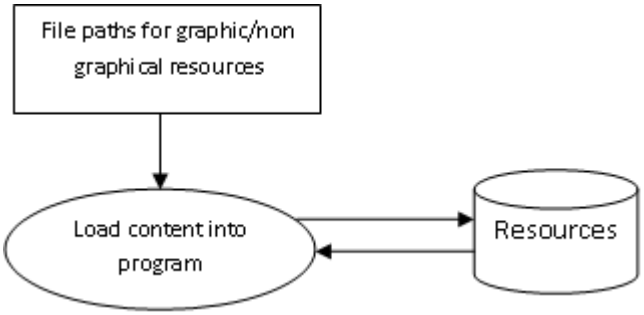


Figure 4 – Low Level Data Flow Diagram

As you can see from Figure 4, this part of the engine simply looks up the file paths of the game resources and loads them into the game.

Game Loop

This final part of the game engine is the "heart" of the whole architecture. The game loop is essentially an infinite loop that will keep executing at a certain rate and executing the code therein. The game architecture makes use of a game loop that makes constant calls to a "Draw" function which will update all of the graphics on the screen whilst the game loop updates the physical object positions, states etc.

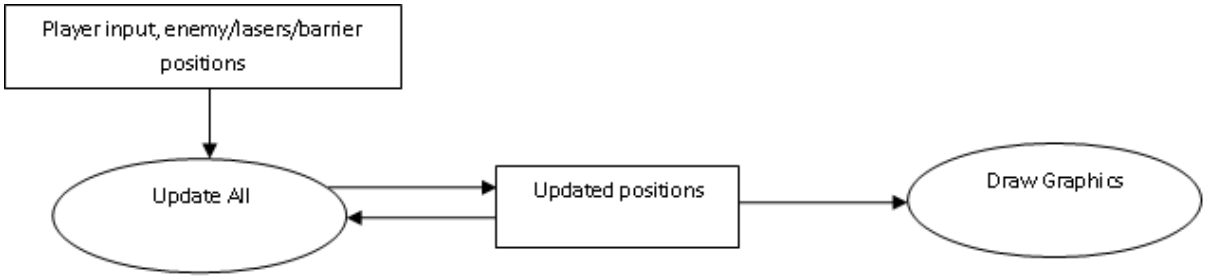


Figure 5 – Low Level Data Flow Diagram

At this stage it is now possible to plan further detailed specifications for each noted object with the overall intention to map the specifications to CRC cards which will be extremely useful when it comes to the physical implementation of the game. As noted in the Interim report the specification for each object is as follows:

- Player
  - Move horizontally at the bottom of the screen;
  - Fire Lasers;
  - Have a finite number of lives;
  - Have a score associated with the number and type of enemy destroyed.
- Enemies
  - Move as a group from one side of the screen to the other;
  - Have a set score associated with each enemy;
  - Only enemies who have the line of sight of the player can fire lasers.
- Barriers
  - Absorb laser damage from both the player and the enemies;
  - Have a finite amount of damage it can take before being destroyed.
- Screens
  - Be intuitive and effectively display all critical information (health & score).
- Lasers
  - Move up or down the screen depending on who fired it.

From here it is relatively simple to insert these objects into CRC card format. The next step is to try and formulate their behaviour as stated in the specification as high level methods and assign basic variables to each object.

CRC Cards

Following on from the specification, the next focus is to propose viable classes for each of the objects outlined previously. An effective way to do this is with the use of CRC cards. This will allow for effectively "programming" classes along with their associated variables and methods in order to gain a better understanding of the bigger picture.

Class: Player

This class is probably one of the most self explanatory with regards to what it needs to do and what variables are required. The basic attributes for this class are as follows:

Attributes:	Type:	Description:
Lives	Int	Number of lives for the player
Score	Int	Current score for the player
Position	Vector	Coordinate of the top left corner of the player rectangle
Movement Speed	Float	The movement speed for the player
Shield	Boolean	Dictates whether the players shield is activated or not
Dual Lasers	Boolean	Dictates whether the players dual lasers are activated or not
Player Rectangle	Rectangle	The bounding box (with aspect ratio applied) for the player texture to be drawn
Texture	Texture2D	The graphical texture for the player
Active	Boolean	Represents if the player is alive/dead

The basic methods that will be associated with a player class are also reasonably self explanatory given the specification. One method that has been added that is less explanatory however the "Draw" method is. This method has been added in anticipation of the requirement for the game to draw the player object to the screen.

Method:	Parsing:	Description:
Void Initialise()	Player textures, Viewport	Initialise the player object with the required textures
Void MoveLeft()	n/a	Update the players position with regards to its movement speed
Void MoveRight()	n/a	Update the players position with regards to its movement speed
Void Draw()	SpriteBatch	Draw the player texture given its position
Void LoseLife	n/a	Decrement the players current remaining lives

Class: Enemy (Master)

Sticking with the original Space Invaders game it is clearly noted that in each row (or every other) there is a different type of enemy of which is worth more points (working from bottom up). Therefore at the design stage it is necessary to prepare a suitable architecture for managing several types of an enemy. A relatively easy way to do this is to use polymorphism and create a super class for an enemy object which will contain all basic variables and behaviours. The architecture for this can be illustrated diagrammatically like so:

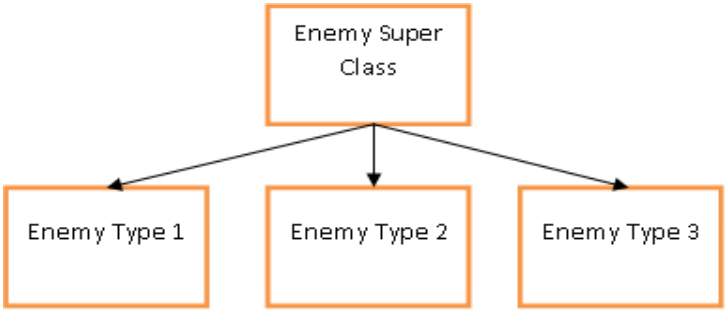


Figure 6 – High Level Class Diagram

With this in mind the attributes for the enemy super class are as follows:

Attribute:	Type:	Description:
Value	Int	The value added to the players score if that enemy is destroyed
Position	Vector	The coordinate of the top left hand corner of the enemy rectangle
State	Enum	The various states an enemy object can be in.
PreviousState	Enum	Represents the previous state the enemy was in.
Speed	Float	Dictates the speed of the enemy
Enemy Rectangle	Rectangle	The bounding box (with aspect ratio applied) for the enemy texture to be drawn
Texture	Texture2D	The graphical texture for the enemy

These attributes will be applied to every sub class of enemy that is created. The next step is to create the methods that each enemy will be likely to use. Keeping in mind the behaviour of the enemies in the original Space Invaders it is noted that each row of enemies move together in a stuttering fashion. Therefore a method will be added to the super class that will be able to update an enemy's position on the screen.

Method:	Parsing:	Description:
Void Initialise()	Enemy Texture, Viewport	Initialise the enemy object with its texture and position
Void MoveLeft()	n/a	Move the enemy by updating its rectangle position
Void MoveRight()	n/a	Move the enemy by updating its rectangle position
Void MoveDown()	n/a	Move the enemy by updating its rectangle position
Void Draw()	SpriteBatch	Draw the enemy texture given its position

Class Common Enemy: Enemy

- Attributes and Methods same as super.
- Value attribute modified to low score.
- Overridden draw method.

Class Uncommon Enemy: Enemy

- Attributes and Methods same as super.
- Value attribute modified to medium score.
- Overridden draw method.

Class Very Uncommon Enemy: Enemy

- Attributes and Methods same as super.

- Value attribute modified to high score.
- Overridden Draw Method.

Class Mystery Enemy: Enemy

The mystery enemy will require a few extra methods and attributes to any of the main enemies found in the game due to its difference in behaviour. Namely an extra method that will be added is "Generate Speed". This will generate a random speed in which the enemy will travel across the screen. The purpose of this is that the faster the enemy is moving, the more points it will be worth upon destruction. However this fact will not be known to the gamer, the point of this is for the player to make the association of the above and then prioritise whether to try and shoot it down or not. The method details in CRC card form can be found below:

Method:	Parsing:	Description:
Void GenerateSpeed	n/a	Generate a random speed for the mystery enemy and calculate its value based on its speed.

Class Laser

The laser classes are by design one of the simplest classes that will be implemented in the game due to its simplistic behaviour in the game. The attributes required for a laser object is the following:

Attribute:	Type:	Description:
Speed	Float	The speed of the laser
Texture	Texture2D	The laser graphic
Position	Vector	The graphical texture of the laser
Active	Boolean	Represents whether the laser is active or if it has been destroyed
Laser Rectangle	Rectangle	The bounding box (with aspect ratio applied) for the laser texture

As per the initial specification, if a laser is fired by the player then it is to travel up the screen until it collides with an object or reaches out of bounds.

Method:	Parsing:	Description:
Void Initialise	Laser Texture, Viewport	Initialise the laser object with its texture and position
Void UpdatePosition	n/a	Update the position of the laser so it travels up the screen
Void Draw	SpriteBatch	Draw the laser graphic to the screen

Class Enemy Laser

The enemy laser class is very similar to its player laser counterpart if not exactly. The only difference between them is their behaviour namely, that a laser fired by an enemy will travel down the screen. Therefore all of the enemy laser attributes will be kept the same as shown previously.

Attribute:	Type:	Description:
Speed	Float	The speed of the laser
Texture	Texture2D	The laser graphic
Position	Vector	The graphical texture of the laser
Active	Boolean	Represents whether the laser is active or if it has been destroyed
Laser Rectangle	Rectangle	The bounding box (with aspect ratio applied) for the laser texture

Note that the following methods will also appear exactly the same as the previous laser class as CRC cards do not delve into too much detail.

Method:	Parsing:	Description:
Void InitialiseLaser	Laser Texture, Viewport	Initialise the laser object with its texture and position
Void UpdatePosition	n/a	Update the position of the laser so it travels down the screen
Void Draw	SpriteBatch	Draw the laser graphic to the screen

Class Barriers

The barriers are by design to protect the player during the first couple of rounds of game play, or for more experienced players to reach the highest possible round before dying. With this in mind a barrier will be made up of several “barrier blocks”. The barrier class will be used to create barrier block instances in game. Each barrier block will have the following attributes:

Attribute:	Type:	Description:
Barrier Texture	Texture2D	The graphical texture of the barrier
Barrier Rectangle	Rectangle	The bounding box (with aspect ratio applied) of the barrier texture
Health	Int	The amount of health per barrier block
Position	Vector	The position of the barrier relative to the top left hand corner of the rectangle
Active	Boolean	Represents whether the barrier block is active or not

As each barrier block is stationary in game, the number of methods required will be at a minimum. The only bespoke method required is one to reduce the health of the barrier when it is hit.

Method:	Parsing:	Description:
Void Initialise()	Texture, Viewport	Initialise the barrier object with its texture and position
Void LoseLife()	n/a	Decrement the current remaining life of the barrier
Void Draw	n/a	Draw the barrier object relative to its position

Class Main Menu Screen

The next section of classes is dedicated to the numerous "screens" that will be used in the game. A screen represents the current state of the game and what is to be shown on the screen given that state. Each screen will share common attributes that will include vectors for text positions etc. The attributes for the main menu screen are as follows:

Attribute:	Type:	Description:
Text Positions	Vector	The coordinates for all of the text to be drawn to screen
Menu Position	Int	Represent the option the user currently has selected on the screen
Fonts	SpriteFont	The different fonts used in the main menu
Sub Menu Active	Boolean	Represents whether or not to expand the sub menu

Each screen will also have its unique set of methods based upon how the user will navigate through the textual options shown on the screen. As this is the main menu screen the user will have to do a fair amount of navigation in order to select the option they like. With this in mind the methods required are as follows:

Method:	Parsing:	Description:
Void Initialise()	Texture, Viewport	Initialise the main menu object with its texture and position
Int Menu Up	Current menu position	Update the menu
Int Menu Down	Current menu position	Update the menu
Void Expand Sub Menu	n/a	Expand the sub menu
Void Draw	n/a	Draw the main menu
Void Change Font()	Menu position, smaller font, bigger font	Change the font size and colour of the highlighted option
Void Draw Sub Menu()	n/a	Draw the main menu along with the expanded sub menu

As you can see from the table above a very basic option manager will be implemented which will work by recording and manipulating a single integer value to reflect the current

option that is highlighted by the user. In order to make the user's choice obvious to the user a "Change Font" method will be implemented which will increase the size of the font that has been highlighted. There is also the possibility to change the colour of the selected option in order to emphasise this.

Class High Scores Screen

The high scores screen is by design to do one thing, namely to effectively portray the highest scores achieved by previous players and the high score achieved by the user (if it is high enough). As high scores will need to be scored in non volatile memory (as to avoid deletion between running the game) a method will need to implemented that will read and write the high score information to a file. With this in mind the following attributes will be declared:

Attribute:	Type:	Description:
Text Positions	Vector	The coordinates for all of the text to be drawn to screen
Score Positions	Vector	The coordinates for all of the scores to be drawn to screen
Fonts	SpriteFont	The different fonts used in the main menu
Scores	Int[]	Stores all of the highest scores
FilePath	String	The full path location of the high score file

As you can see, a string array will be used to store the characters read from the file. There is also a need to store the intended directory of the file in a string format so a method knows where to access the file to read from. The last method that will be required is to update the high scores file given the score the current user has achieved.

Method:	Parsing:	Description:
Void Initialise()	Texture, Viewport	Initialise the high scores object with text positions
Void CreateFile()	n/a	Attempts to locate an existing high scores file and if it fails it creates a new file
Void OpenFile()	n/a	Attempts to open the high scores file and read its contents
Void UpdateScores()	NewScore, PlayerName	Updates the high scores based on the players score
Void Draw()	SpriteBatch, SpriteFont	Draws the high score screen

Class Paused Screen

This screen class will be used to represent what is drawn to the screen when the user decides to pause the game. This screen will be one of the simplest by design given that it will only display to the user:

- That the game has been paused;
- How to resume the game;
- How to quit the game and return to the main menu.

Therefore the only attributes required will be those to represent the position of the text on the screen like so:

Attribute:	Type:	Description:
Text Positions	Vector	The coordinates for all of the text to be drawn to screen
Fonts	SpriteFont	The different fonts used in the paused screen

Method:	Parsing:	Description:
Void Initialise()	Texture, Viewport	Initialise the paused screen object with text positions
Void DrawPausedScreen()	SpriteBatch, SpriteFont	Draws the paused screen

Class Game Over Screen

The game over screen is a necessity in any video game as it not only informs the user that the game has conclusively come to an end but, it can also act as a form of humiliation such that for example, the "game has won" and the player has lost. Again to implement the screen will require the same basic attributes and methods found in the other screen classes.

Attribute:	Type:	Description:
Text Positions	Vector	The coordinates for all of the text to be drawn to screen
Fonts	SpriteFont	The different fonts used in the paused screen
Position	Integer	Portrays the option the user has currently highlighted

As the game will effectively be over, the user will be presented options to either return to the main menu (to start another game) or to exit the game all together. Therefore the class will require methods to handle the user's navigation to these options.

Method:	Parsing:	Description:
Void Initialise()	Texture, Viewport	Initialise the game over object with text positions
Void UpdateLeft()	n/a	Decreases the value of Position if the first option is NOT currently highlighted
Void UpdateRight()	n/a	Increases the value of Position if the second option is NOT currently highlighted
Void DrawScreen()	SpriteBatch, SpriteFont	Draws the game over screen

Class Instructions Screen

The instructions screen will display the objectives of the game, how many points each enemy is worth and for the modern version of the game, what power ups do what. Therefore as this screen will be used to just display text it will contain the following attributes.

Attribute:	Type:	Description:
Text Positions	Vector	The coordinates for all of the text to be drawn to screen
Fonts	SpriteFont	The different fonts used in the instructions screen

As a result of the user not interacting with this screen the only methods required are those to initialise the screen and to draw the screen.

Method:	Parsing:	Description:
Void Initialise()	Texture, Viewport	Initialise the instructions object with text positions
Void DrawScreen()	SpriteBatch, SpriteFont	Draws the instructions screen

Class Play Screen

The play screen class will be used to display the user interface whilst the game is in its "play game" state. This includes items such as: round number, score and number of lives remaining etc. Therefore the attributes are as follows:

Attribute:	Type:	Description:
Text Positions	Vector	The coordinates for all of the text to be drawn to screen
Fonts	SpriteFont	The different fonts used in the paused screen

There will be a number of drawing methods required in this class in order to draw each piece of information to the game screen of which are found below:

Method:	Parsing:	Description:
Void Initialise()	Texture, Viewport	Initialise the play screen object with text positions
Void DrawLives()	SpriteBatch, SpriteFont	Draw the number of lives remaining to the screen
Void DrawScore()	SpriteBatch, SpriteFont	Draw the current score to the screen
Void DrawRound()	SpriteBatch, SpriteFont	Draw the current round number to the screen
Void Draw Border()	SpriteBatch	Draw the border that separates the information from the game play area of the screen

Class Power Up

As per the Interim Report, one of the possible advanced features to be used in the modern version of the game is the use of power ups. A power up can be traditionally found in two forms: its inactivated form and its activated form. Therefore with regards to implementing the power ups I will need a way to represent the power up in each form. Therefore a suitable approach is to implement a power up class that will be used to represent a power up in its inactive state. Whilst in this state the class will determine which kind of power up it is and how it is to move across the screen. Traditionally in video games a power up is a reward for destroying an enemy or completing a specific task. With this in mind the decision was made that when an enemy is killed, there is a certain chance of a power up being dropped from its position and "falling" down the screen in a straight line. Therefore the attributes for class are as follows:

Attribute:	**Type:**	**Description:**
Active	Boolean	Depicts whether the power up object is active or not
Position	Vector	The position of the power up on the screen
PowerUp Rectangle	Rectangle	The bounding box (with aspect ratio applied) for the power up texture
Drop Speed	Float	The speed at which the power up moves down the screen
ShieldPowerUpTexture	Texture 2D	The texture for the shield power up
DualLasersPowerUpTexture	Texture 2D	The texture for the dual lasers power up
MovementPowerUpTexture	Texture 2D	The texture for the movement speed power up

The methods for a power up are relatively similar to that of the enemy lasers, given that they are created at the enemy's position and then move down the screen. Therefore the methods for this class are as follows:

Method:	**Parsing:**	**Description:**
Void InitialisePowerUps()	Shield texture, movement texture, dual lasers texture, viewport	Initialise the power up object along with all of the power up textures
Void DropPowerUp()	Position, power up type	Add a power up to the game screen
Void UpdatePowerUp()	Position, power up type	Update the power ups position so it moves down the screen
Void DrawPowerUp()	SpriteBatch	Draw the power up to the game screen

Class Bayesian Network

As per the Interim Report, one of the main objectives in the modern version of the game is to make the enemies seemingly more "intelligent" with a hope to make the game play more challenging and enjoyable. However without conducting the design of the network itself, at this stage it is very difficult to plan ahead for the implementation of this class. Therefore this class from a design perspective shall be a basic interpretation.

Attribute:	Type:	Description:
Probabilities	Float	Variables used to store the probabilities of an event occurring
Truth Values	Boolean	Describes whether an event has occurred or not

The methods for this class however are reasonably self explanatory given that, in order to create a probability, one must:

- Determine what events have occurred;
- Given their probabilities, calculate the probability of an enemy firing.

Therefore it is reasonable to expect the methods for this class to be similar to the following:

Method:	Parsing:	Description:
Void InitialiseNetwork()	Probabilities of events occurring	Initialises the network and assigns probabilities to their respective variables
Void DetermineEvents()	Truth values of certain events	Given the games current state, determine what events have occurred and store them in their Boolean form.
Void CalculateProbability()	All of the above	Given

Main Menu Screen User Interface

The main menu screen design is extremely simple and is only used to allow the user to navigate through the options and make their selections. It will provide the user with the opportunity to read through the instructions and the controls before starting a game.

In order to illustrate what option the user has currently highlighted the menu will change the size and colour of the font of the option to reflect this.

Figure 8 – Main Menu User Interface

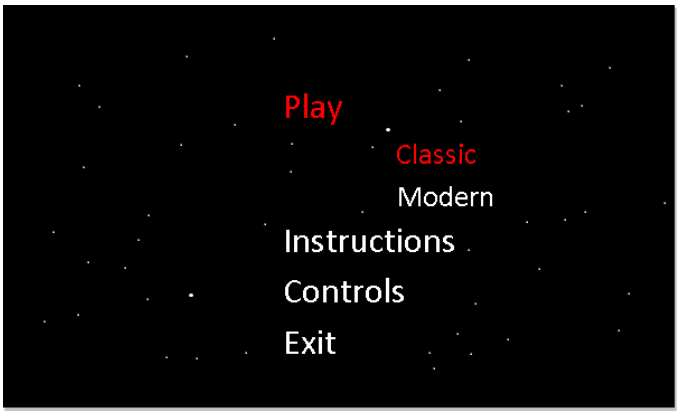


Figure 7 – Expanded Menu User Interface

When the user selects the play option, the menu will expand to reveal sub menu options. This is where the user can select to play the classic rendition of the game or the modern interpretation of the game. Again the menu will make use of highlighting the current option the user has selected.

Game Screen User Interface

The game screen user interface is not used to draw all of the items to the game screen as each object will have its own draw method. Instead it will be used to draw specific information to the screen for the players benefit.

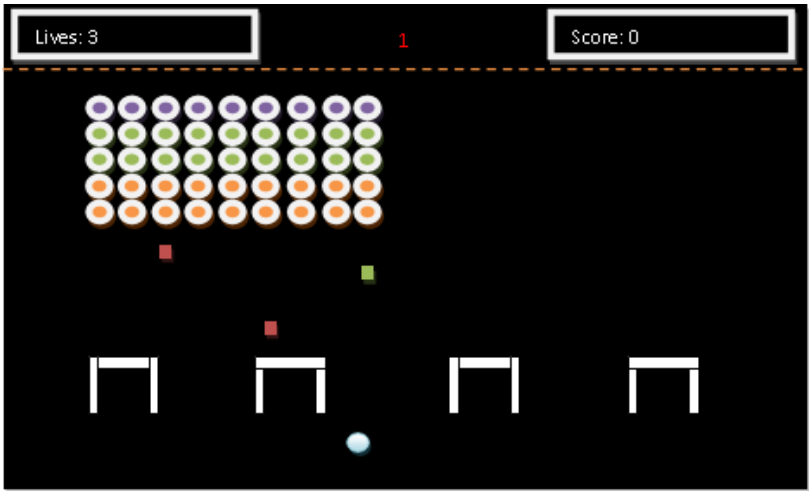


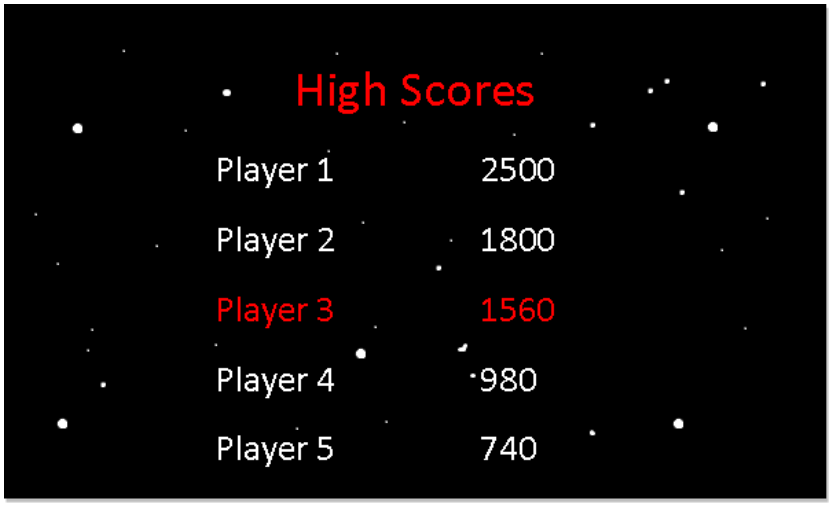
Figure 9 – Game Screen User Interface

The heads up display (HUD) for the player during the game screen must display the following:

- Number of lives remaining;
- Current round number;
- Current score.

High Scores User Interface

This screen will be used solely for displaying the highest scores achieved in a game when the player has lost all of their lives.



Only the top 5 scores will be saved and if the players score is high enough it will be highlighted.

Figure 10 – High Scores User Interface

Control User Interface

The concept behind this screen is that, if the user is playing the game on a PC, then the control screen will be able to detect what controller the player is using (i.e. keyboard or Xbox controller). Therefore two different screens of information must be considered based on the player's choice of controller.

Figure 11 – PC Control Screen User Interface

Figure 12 – Xbox Control Screen User Interface

From figures 11 and 12 it can be seen that the screen layout is very simple and merely illustrates what control does what in game. There is the possibility of adding an image of the Xbox 360 controller and PC keys to better illustrate this point if there is time later in the project.

Instruction Screen Design

The instructions screen is required in almost any game in order to provide a concise overview of the game objectives and what to expect in the game. This is crucial as not all users will be seasoned gamers and be able to naturally adapt to what is happening compared to new comers.

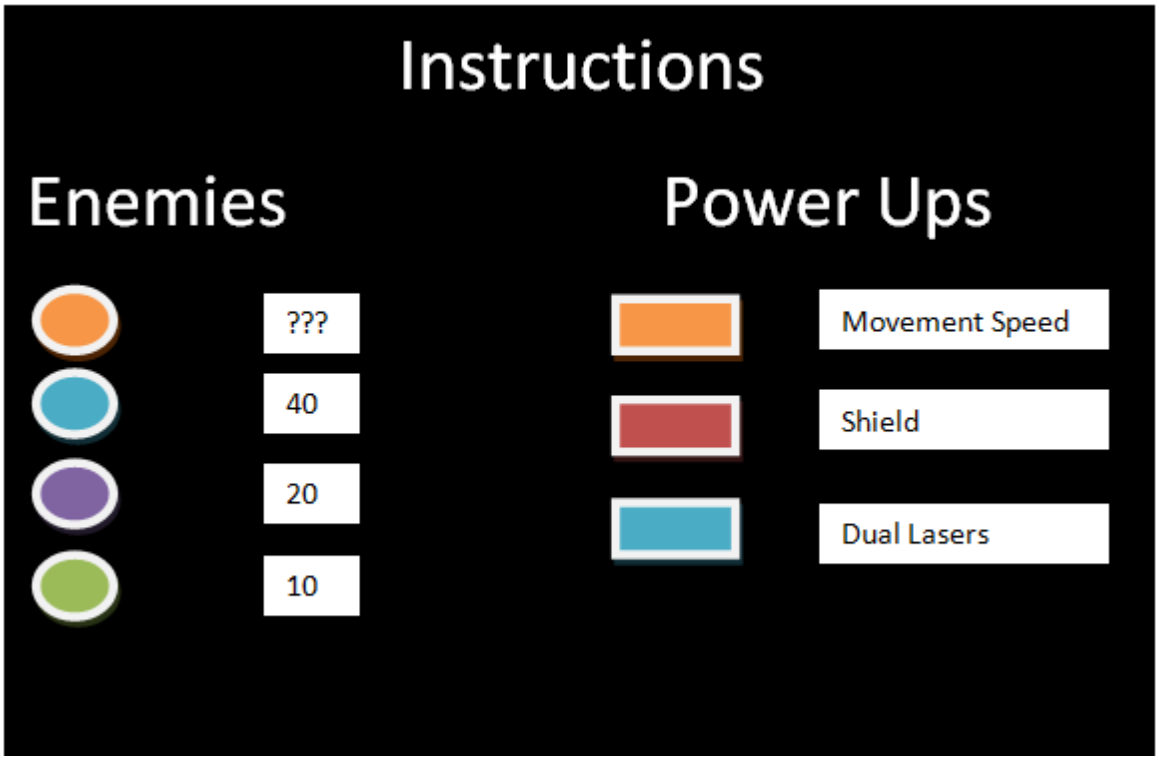


Figure 13 – Instructions Screen User Interface

Figure 13 illustrates that the instructions screen will display all of the possible enemies and power ups that can be encountered in the game play along with a description. It is important not to overcomplicate the display of instructions to the user as to avoid any confusion which could impact the overall gaming experience.

With this in mind the implementation of the same will be constantly checked against the original design. However the design is open to change based on user feedback and their opinions on the instructions clarity and overall layout.

Game Screen Management Flow Chart

As the game has several different screens, these can be treated as "states", this allows for the design of the interactions between each state and the required scenario for a transition to occur. The following diagram illustrates each screen in the game and how the user progresses from one screen to another.

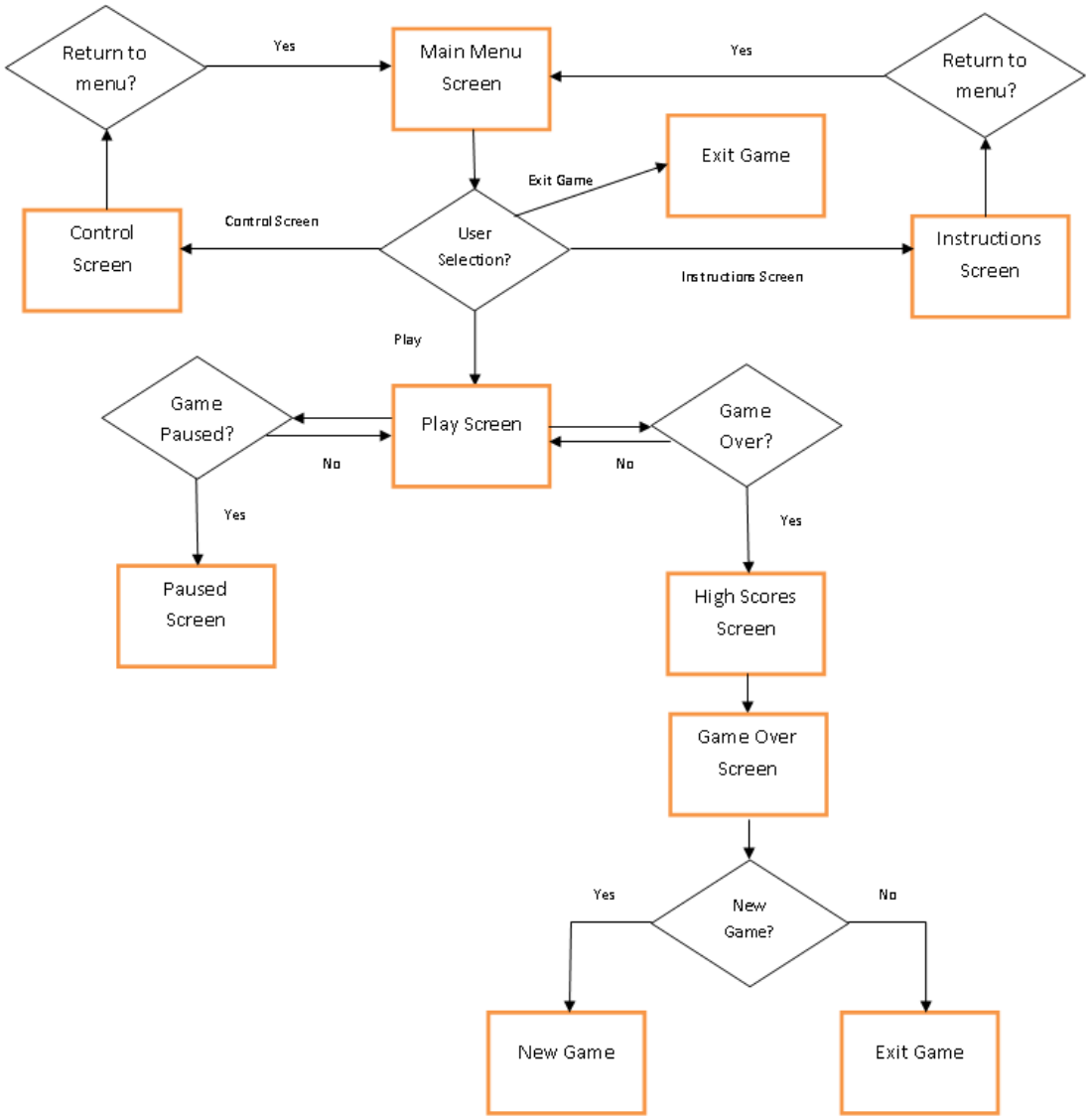


Figure 14 – Game Screen Management Flow Chart

From the flow chart we can determine that:

- The main menu screen acts as the central hub of all other screens and links most of them together.
- The play screen is constantly checked to see if the game has been paused or has finished.
- The transition occurs between the high scores screen and game over screen after an allotted time frame and therefore there is no user interaction at this point.

Player Input Design Considerations

As the game is designed to work on multiple platforms it must be taken into account the variety of different input devices that can be used to interact with the game. For example some different types of input given from using a PC, Windows Phone and an Xbox 360 includes: Mouse, keyboard, hands, gestures, accelerometer, game pad etc. Therefore the screen designs have ensured that no matter what manner of interaction is occurring the player will be able to seamlessly play and enjoy the game to its full potential.

Mobile Phone Play Style Vs Console Controls

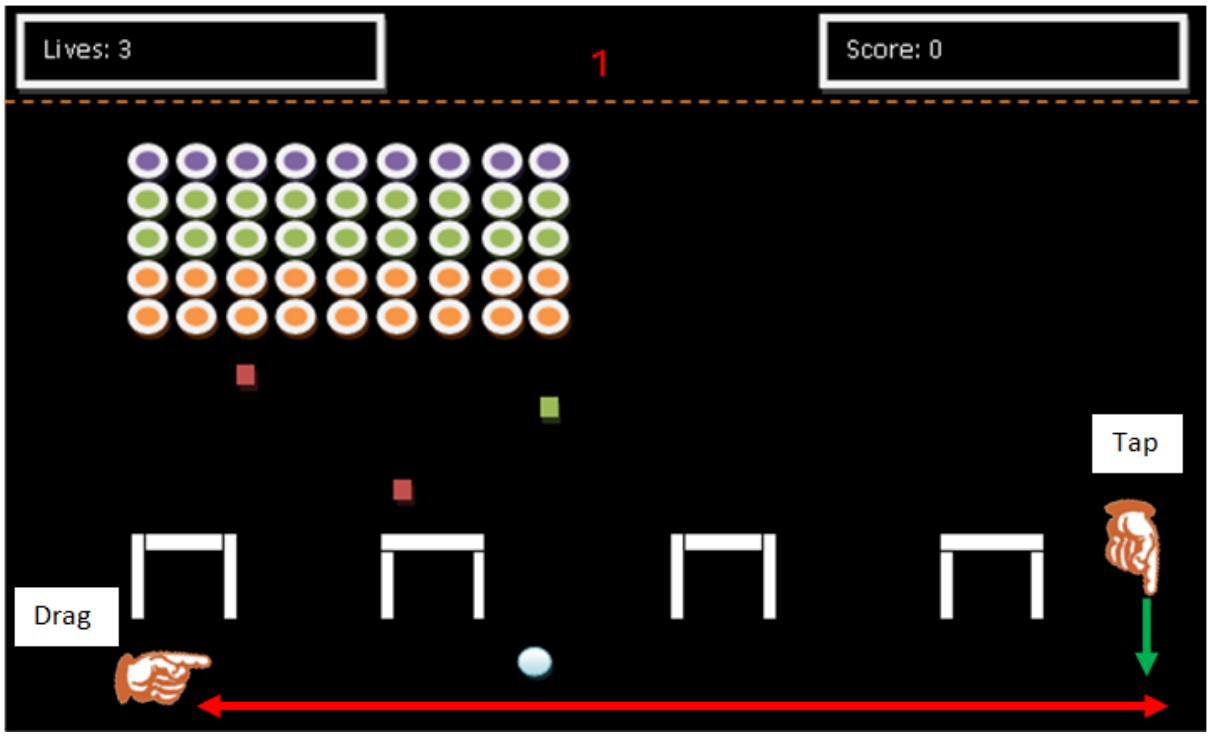
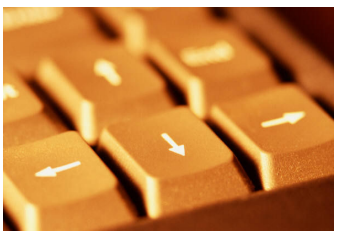


Figure 15 – Mobile Phone Input Considerations

For the mobile version of the game it was decided to use a simple finger input play style. To move the player ship the user merely has to drag their finger across the bottom of the screen and the ship will follow. To fire a laser the user simply has to tap the screen once. The option was available to make use of the accelerometer to control the player movement however, after some consideration it was dropped due to the fact that physically moving a device whilst trying to focus on positions of the enemies and lasers etc may be too strenuous for the player and can cause distress/frustration as opposed to enjoyment.

The PC version of the game will make use of the standard input; i.e. the keyboard. The arrow keys are the most widely used form of input for movement in a game and therefore I have opted to proceed with this option.



The Xbox controller allows for a lot more variety when it comes to input as it is a purpose built, ergonomic game controller. It is also able to bring the game more to life by using its “rumble” feature. This can be used to give the player feedback from the game via the controller as opposed to just looking at the screen. An example layout for the controls of my game can be found on the following page.

Figure 16 – Xbox 360 Control Considerations

Using a Xbox controller will provide the user with a more "arcade" like game play experience due to the use of a joystick to control the player movement.

Therefore having such a variety of controls for this game provides the user the opportunity to experience the same game on different platforms which was not the case for the original Space Invaders game.

Collision Detection

Collision detection is a major aspect of any game. This is where the game uses a collision detection algorithm to determine whether two (or more) objects have collided. For this solution a bounding box approach will be used, for which the algorithm is as follows:

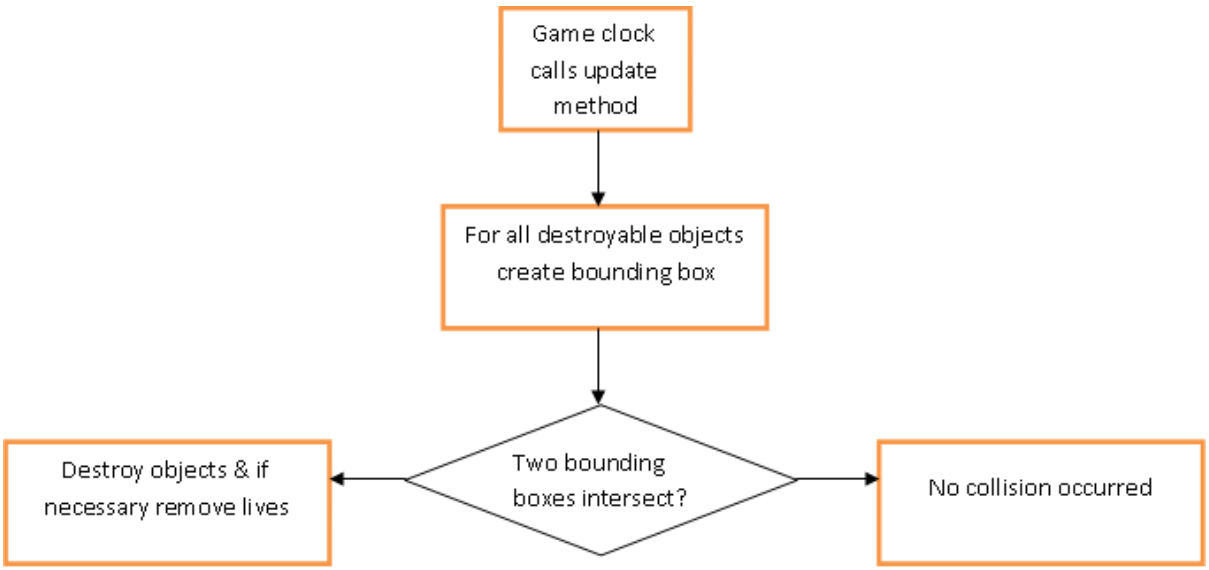


Figure 17 – Collision Detection Algorithm Flowchart

The approach of implementing this algorithm is relatively simple. As shown in Figure17 above, the algorithm will create a bounding box around each object graphic that is drawn to the screen. The algorithm then performs a simple check to see if any of the coordinates along the border of the bounding box has intersected with that of another. This can be easily expressed using an illustrative example like so:

Stage 1 – Creating bounding box

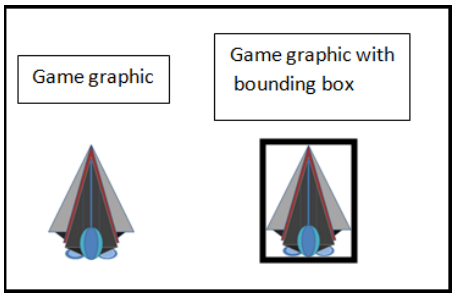


Figure 18 – Bounding a game graphic in a rectangle

Stage 2 – Check for intersections

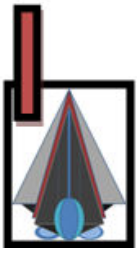


Figure 19 – Collision between two bounding rectangles

Therefore this approach provides an extremely simple solution to collision detection which is also not computationally expensive. Technically this implementation will result in O^n computation time as a collision detection will occur once for each object on the screen.

However this method is not without its draw backs. It is easily noticed from Figure 19 that, it is easy for the bounding boxes to collide whereas the actual graphics contained within them have not.

Therefore it may be an issue if a collision does not look believable to the player in game. There are however, a number of alternative approaches to make the player "believe" a collision has occurred.

Accurate Collision Detection

In case the collisions in the game do not look believable to the player there are a couple of tweaks to the algorithm that can be made to try and compensate for this. One simple option is to merely increase the speed of the missile moving across the screen, thereby making it more difficult for a human to be able to see the absolute point of contact.

Another, more feasible option would be to manipulate the collision detection points in the rectangle in such a way that, if anything hit that particular point it would be an undeniable strike as opposed to a near miss. This can be done by creating a rectangle that is slightly smaller than the game graphic itself, like so:

Figure 20 – Narrower bounding box

By creating a smaller contact area this massively reduces the possibility of a laser missing the space ship graphic and still destroying it. This approach can be replicated for the lasers such that, the bounding box would be slightly shorter than the beam itself. This would ensure that a certain proportion of the beam has made contact with something before checking if a collision has occurred.

At this stage in the project development, the foreseeable collisions will be between the following:

- Player – Enemy
- Player – Enemy laser
- Player – Barrier
- Player – Power up
- Player laser – Enemy
- Player laser – Barrier
- Player laser – Power up

Barrier Design Considerations

Keeping in mind the above discussion regarding collision detection (i.e. using bounding rectangles) the barriers design has made full use of bounding boxes.

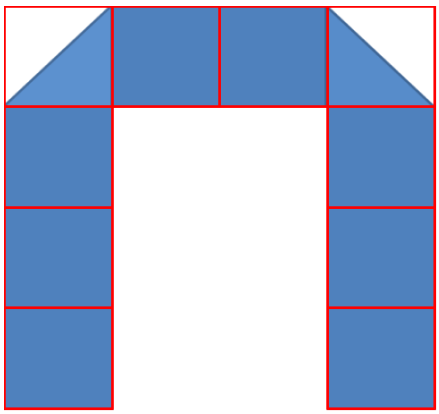


Figure 21 shows how each barrier is made up of several barrier blocks. It is important to note that the "Barrier" class will not actually control a barrier in its entirety. In fact it will represent each barrier block individually. Therefore each barrier block will have health and bounding collision box.

Figure 21 – Barrier Design

Aspect Ratios

One of the challenges to any game is getting it to look the same on a number of different screens. This is doubly so due to the nature of the project covering a large range in screen sizes roughly between 2 inches and anything above. Therefore it is imperative that all textures used in the game are resized with proportion to the size of the screen.

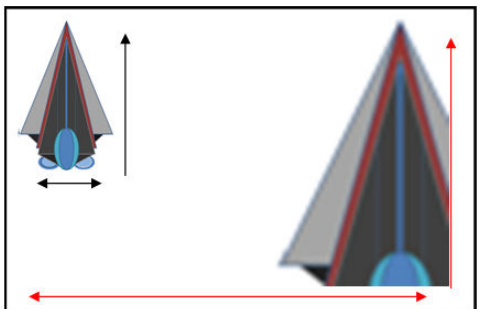


Figure 22 – Game graphics with no aspect ratios applied

As shown in Figure 22, if the physical image file is drawn to the game screen it can either be far too small or too big for the screen itself. Therefore to ensure that this does not happen there is a simple calculation that can be done to create a bounding box for the image to be

drawn in. This box will be proportional to the size of the screen and will scale the image depending on the scaling factor specified in the program itself.

The following variables are required to recreate a bounding rectangle:

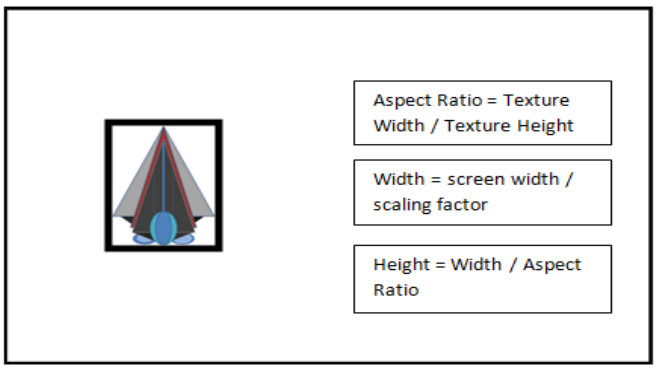


Figure 23 – Aspect Ratio Equation and variables

- Vector
- Width
- Height

Figure 23 represents how to calculate the bounding box for graphic and how it is scaled down.

Bayesian Network

In order to implement a probabilistic Bayesian network the design the network must be done first and take into account the dependencies and probabilities. For the modern version of the game it is my intention to make the enemies "smarter" with their choice of when to fire a laser. As discussed in the interim report, the enemies in the original game merely fired at random and there was no real justification for them to fire. Therefore based on this I have created the following Bayesian network for firing a laser, taking into account and variables that would affect the probability of firing occurring.

In order to do this we need to place ourselves in the enemy's shoes and note what would motivate them to fire at something. As a result the following network has been constructed.

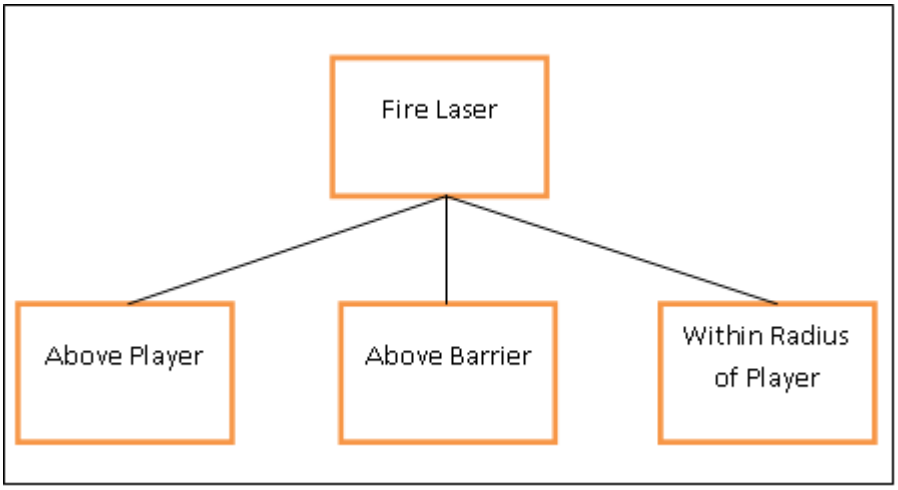


Figure 24 – Bayesian Network Design

From Figure 24 above it is shown that the motivation for a human playing as an enemy is to strategically fire at something with value. This is where firing when you are above the player or a barrier is of particular interest, as the result will yield in a disadvantage to the player that will ultimately end in their demise. However it is also noted that when playing against another player, people would try and predict their opponent's movements and fire a laser into their path as opposed to directly above them. With this in mind another node has been

added to the network that will consider the probability of firing a laser given they are within a specific radius of the player's current position. This will allow for an enemy that is a short distance away from the player to fire a laser that would hopefully intercept the player given their current path.

The network has been designed with a top down approach as to minimise computation. For example, were the network to be flipped upside down it would result in the following computation:

P(1|2,3,4) . P(2) . P(3) . P(4)

Whereas with a top down network the computation is spread far more evenly:

P(1) . P(2|1) . P(3|1) . P(4|1)

Probabilities and Truth Table

With the network constructed the next step is to allocate probabilities to each possible occurrence of firing a laser given something has happened (e.g. being above a player). In order to do this truth tables have been drawn with probability values allocated accordingly.

Firstly a base probability will be allocated for the enemies to fire. This will ensure that an enemy will have a chance to fire even if one of the child nodes are not satisfied. Therefore the base probability of an enemy firing is: P(F) = 0.01. This low value will ensure that at the start of the game the enemies will be far less aggressive and will fire infrequently. As the game progresses this probability will be increased on a linear scale (i.e. P(F) *= round number^2)

Probability of being above the player (P(Ap)) given that a laser has been fired

True	False
0.5	0.25

Probability of being above a barrier (P(Ab)) given that a laser has been fired

True	False
0.3	0.25

Probability of being within a radius (P(Wr)) of the player given that a laser has been fired

True	False
0.4	0.25

Illustrative Example

In order to see how the network will truly affect how the enemies will attack a player, the following diagram represents a game state along with which nodes of the Bayesian network are satisfied (i.e. above the player, above a barrier and within radius of player).

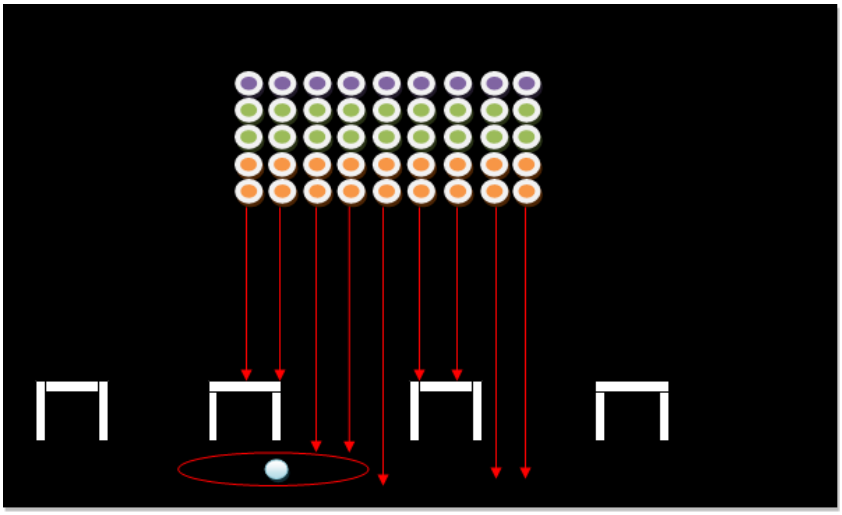


Figure 25 – Game play scenario

The following diagram will now illustrate the probability assigned the each enemy which reflects the probability of each one firing for the first round of the game.

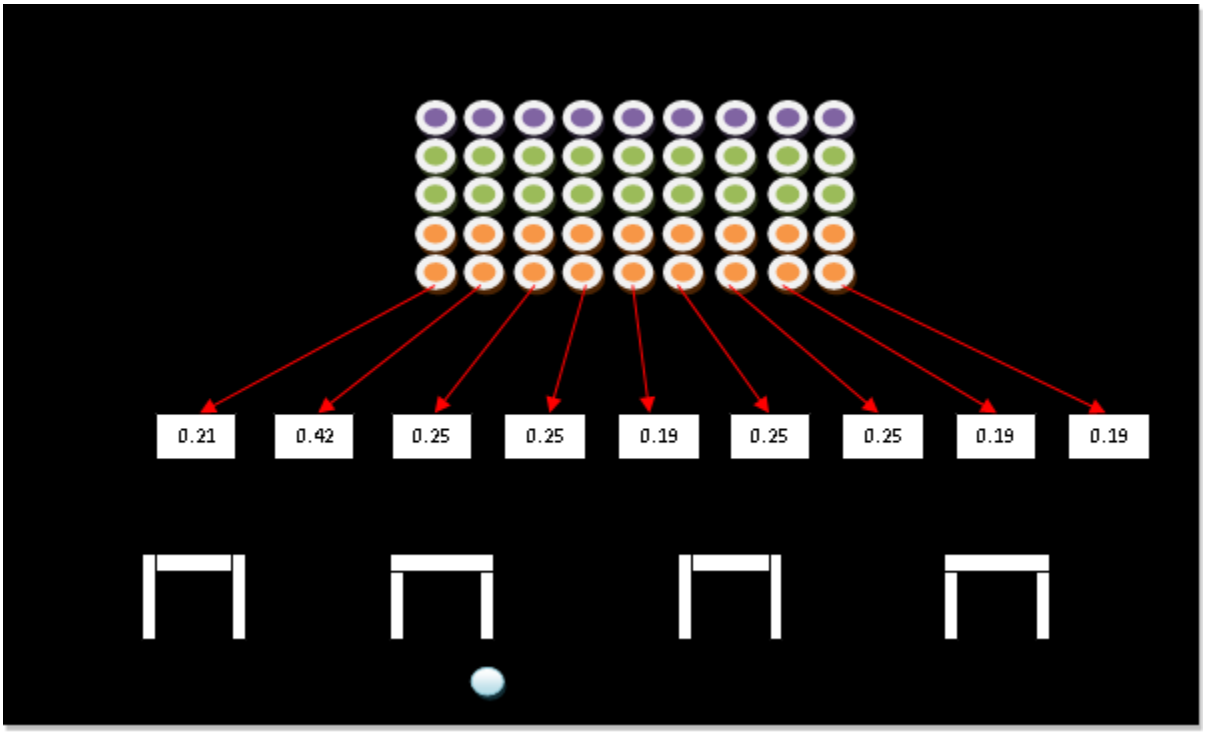


Figure 26 – Probabilities of enemies firing laser

Figure 26 shows the enemies which are either above a barrier, above a player, within the player's radius or a combination of both, have a noticeably higher probability than those who are above nothing at all. However it is important to note that the enemies that are above nothing and satisfy none of the nodes in the Bayesian network are still assigned probabilities which will grant them the opportunity to fire.

The base probability of an enemy to fire is scaled linearly with each round and will therefore gradually increase all of the values found in the above diagram, resulting in a far more aggressive enemy pack. This will account for the scaling difficulty that is found in the original Space Invaders game.

Enemy A.I States

At this stage in the design it is worth considering all the different enemy states and how one state can move to another. As discussed in the interim report the intention is to give each enemy a state and they will then use these states to allow the enemies to change their behaviour.

State Diagram

Below is a state diagram illustrating all possible states of an enemy as well as the different possible states it can transition to.

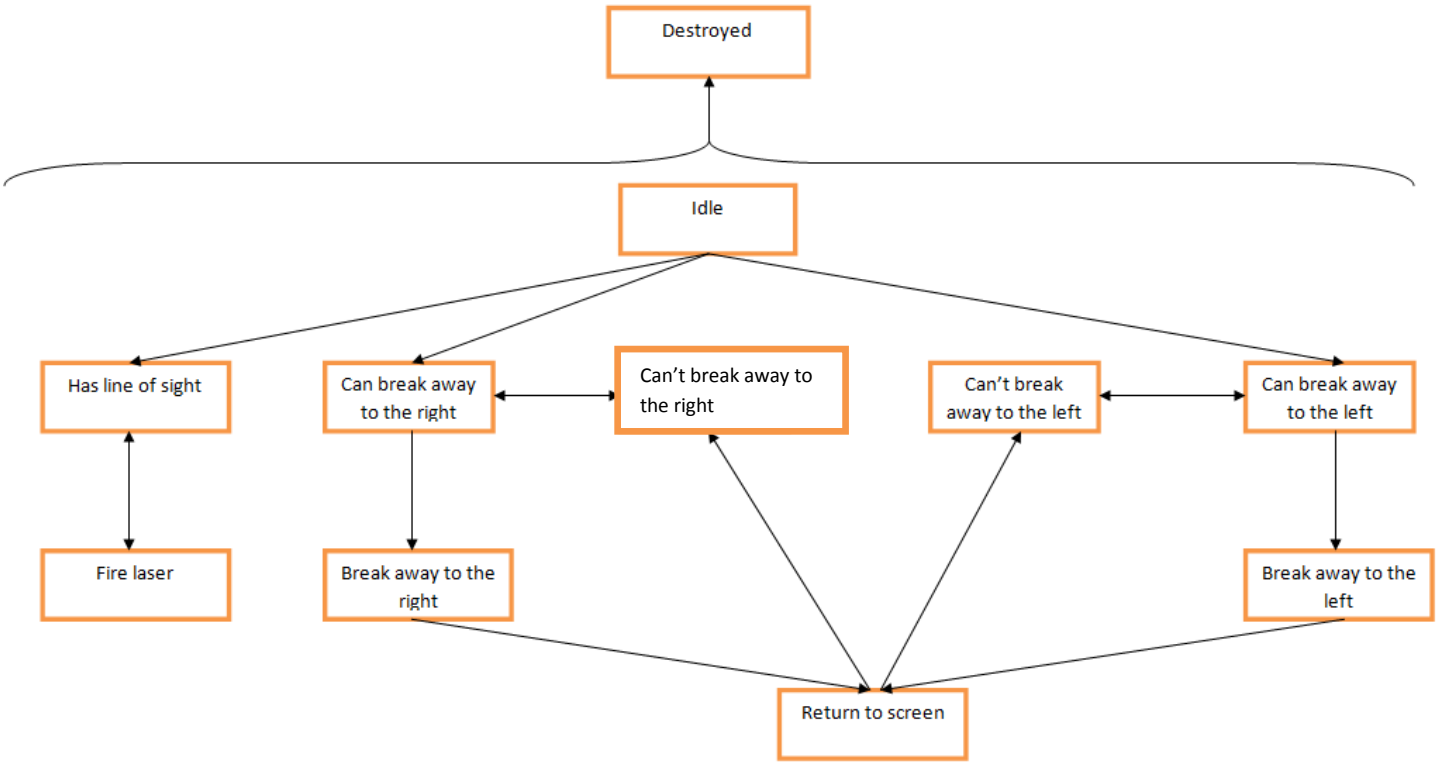


Figure 27 – Enemy State Diagram

Figure 27 illustrates that:

- Each state can make the transition to the destroyed state;
- Each enemy starts out with the base state "Idle" before being allocated something else;
- There is a loop (or cycle) present in the diagram namely that an enemy can transition through the following:
  - Can break away to the right
  - Break away to the right
  - Return to screen
  - Can't break away to the right

This loop exists for the alternate "can break away to the left" scenario.

State Management

It is important to be aware of under what conditions can an enemy make a transition from one state to another. Therefore a list of conditions that must be satisfied in order to allow the transition from one state to another has been created, of which can be found in "Appendix A".

Break Away Enemies

One of the aspects to be implemented in the modern version of the game is the ability for an enemy to break away from the main pack and follow a curved path in an effort to collide with the player or a barrier and cause damage to it. This could be implemented using absolute positioning and a curve could quite easily be generated in this way. However this would be ineffective due to the multitude of screen sizes and therefore an alternative solution is to implement what is known as a "Bezier curve".

A Bezier curve works by considering at least two points and then recursively finds a point between them generating a smooth curve. In this case the cubic formula is most appropriate:

$$B(t) = (1-t)^3P0 + 3(1-t)^2tP1 + 3(1-t)t^2P2 + t^3P3$$

Where t = time and P represents the coordinate points respectively.

The formula requires four coordinate values, these are:

- Key points
  - These are the start and end positions of the curve.
- Control points
  - These are the intermediate nodes in which to calculate and generate a curve.

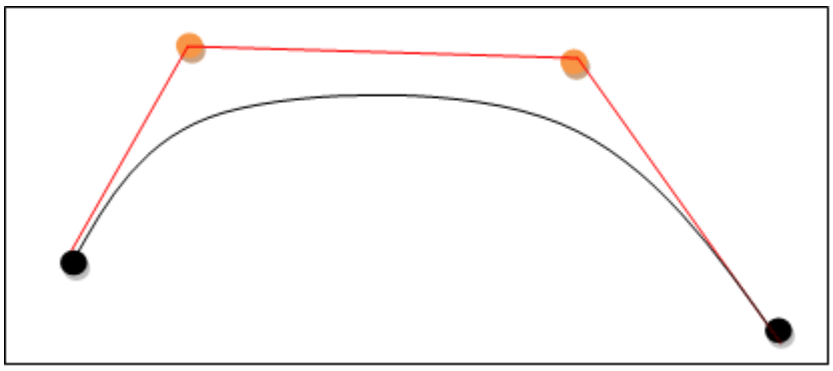


Figure 28 – Design of a Bezier curve

The algorithm to produce this curve is as follows:

For time t, t < 1, t += 0.1

Generate curve point using equation listed above()

This example algorithm will return a curve with a total of 10 points. The lower the increment or the higher the threshold, will result in more curve points and will produce a smoother curve.

Game Application

This curve can be applied to enemies who are breaking away like so:

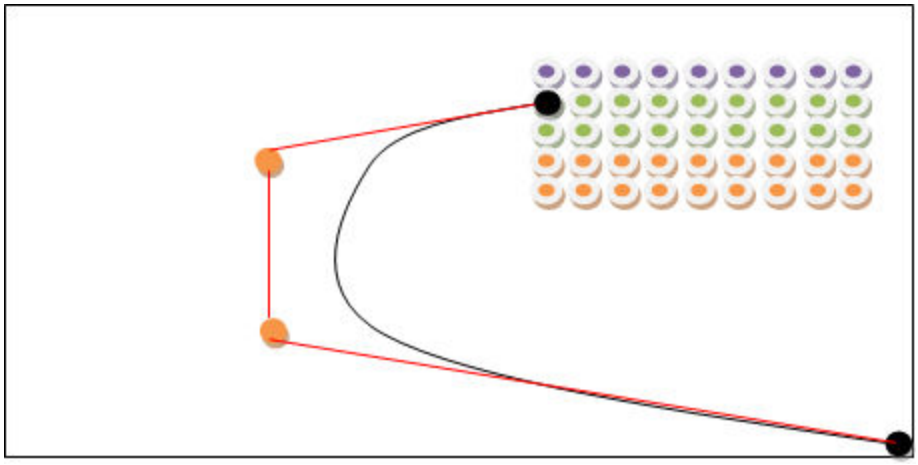


Figure 29 – Game application of a Bezier curve

Figure 29 above represents how the four coordinate points would be declared:

- The key points are :
  - Current position of the enemy
  - Given the direction of the curve, either the bottom left/right hand corner
- The control points are:
  - Specified horizontal and vertical distances from each control point.

In order to use the curve in the game a List will be used to store each coordinate value as the algorithm produces them. From there an enemy can simply loop through the list and update its position accordingly.

Managing Break Away Enemies

With enemies breaking away it is important to consider how to manage when and where they are "allowed" to break away. Due to personal preference, it is unfavourable for the enemies to break away and go out of bounds of the screen. This is because, from a players point of view it would be easier to keep track of every enemies position on screen in order to get a more satisfying and challenging experience. With this in mind we can make use of the Bezier curve algorithm in an iterative loop to find the breakaway threshold coordinates. The proposed algorithm for this is as follows:

Find left most break away threshold()

    While threshold > left side of screen

    GenerateCurveLeft() – This will add all of the curve points to the list

        If current smallest value > curve value

            Current smallest value = curve value[current position]

    End While

As a result of the algorithm this will produce the following scenario:

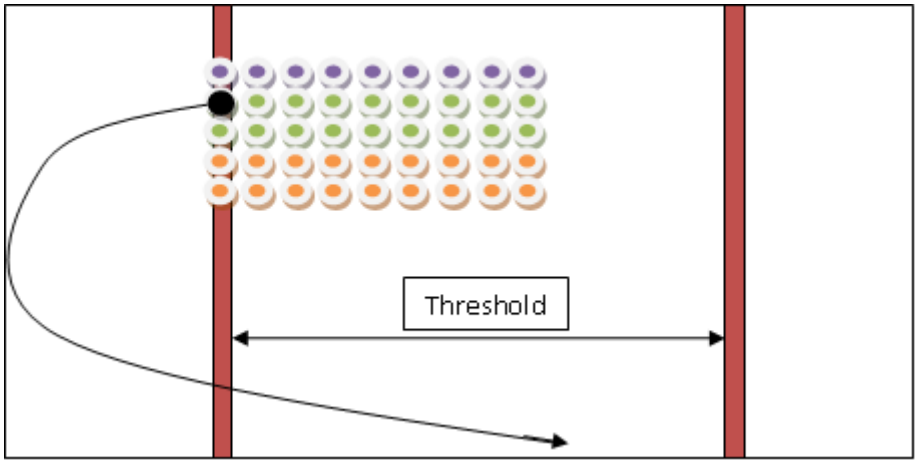


Figure 30 – Break away thresholds

From Figure 30 we can see that the algorithm will find the minimum and maximum possible X coordinates on the screen in which the curve will remain within the bounds of the screen. This algorithm will run before playing commences to avoid any lag time produced that may hinder the smooth running of game play.

Updating Firing Enemies

One of the flaws of the original Space Invaders game was that enemies could fire through other enemies in front of them. In this version of the game it will attempt to rectify this and to allow only the enemies with direct line of sight of the player (not including the barriers) the ability to fire. There are several approaches to manage which enemies do have line of sight to be considered

Enemy Block Jagged Array

A possible solution to managing which enemy has line of sight is to implement what is technically known as a jagged array. A jagged array can be considered as a two dimensional array. This suits the situation at hand as the enemy group is literally a two dimensional matrix.

The idea for this solution is to construct a 10 x 5 dimensional array and assign each number a value (pointer) for the next enemy that is to be fired. This in affect is a link list that will be used to track which enemy is next to fire upon its death.

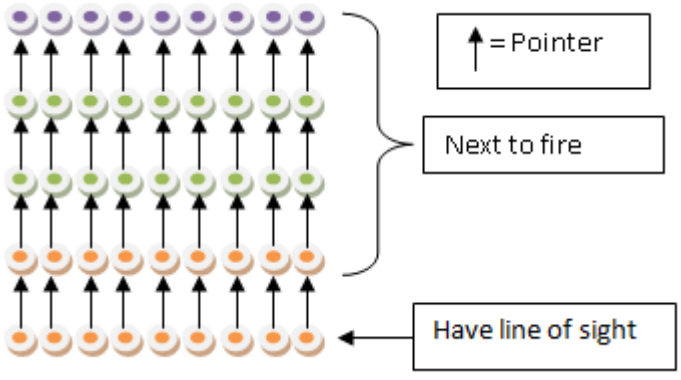


Figure 31 – Enemy firing pointers

Update Firing Enemy Algorithm

When an enemy is destroyed and its state was either “Fire Laser” or “Has Line of Sight” then the program must find that enemies pointer in order to know which enemy is next in line to have “line of sight”. When the game starts, the program will have the following knowledge with regards to the enemy pack. Please note that the table on the right hand side represents the pointer associated with each enemy.

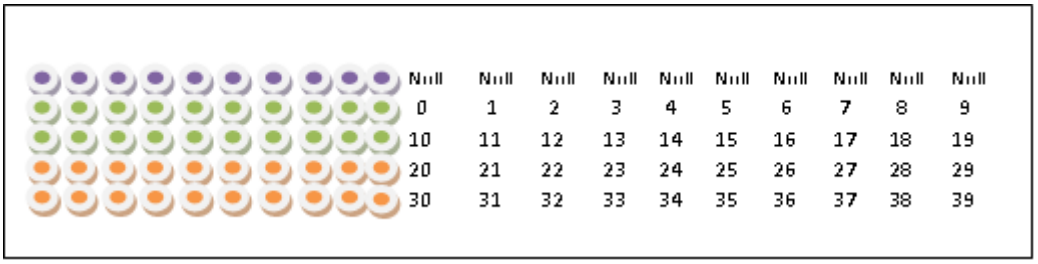


Figure 32 – Assignment of pointers

The next illustration is an example of the enemy pack after several have been destroyed and reflects to condition of the pointers thereafter.

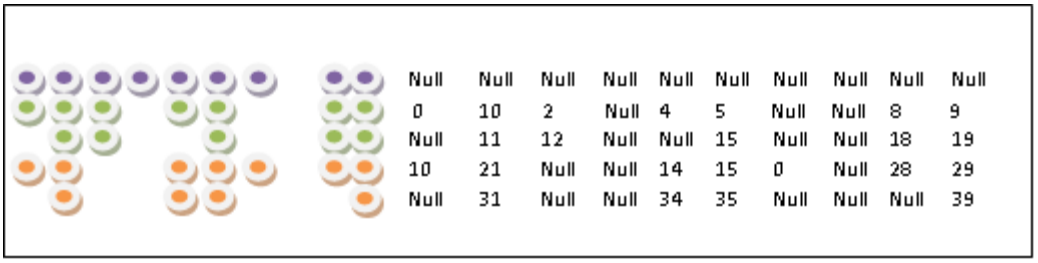


Figure 33 – Game play scenario with updated pointers

Therefore the algorithm to manage the pointers is relatively simple and is shown in the pseudo code below:

If the enemy was firing upon death then

Access its pointer

Assign the enemy number found in the pointer to have line of sight

Else

Access its pointer and store it in a temporary variable

Find the enemy that points to the deceased one

Make its pointer equal to the temporary variable

Managing Enemy Movement

As the enemies move as one in the game, it is necessary to create an algorithm that manages when they change direction and move down the screen. One approach is to track the top left and top right enemies in the pack and monitor when they reach the borders of the screen. When this occurs they will all move down the screen slightly and then change their direction.

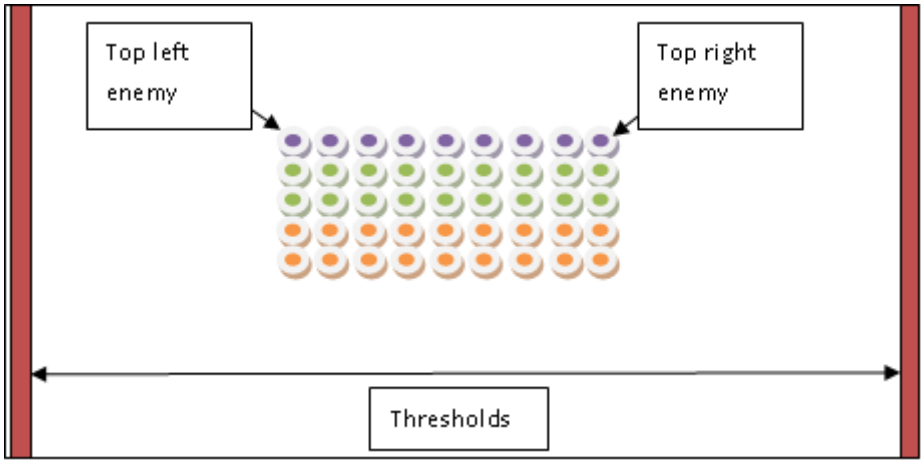


Figure 34 – Assigning screen boundaries

However there is a flaw in this approach, namely that the enemies can be destroyed (killing the top left/right enemy) and therefore the program must be able to find the current left most and right most enemies in order to know when to move down the screen and change direction.

The following illustration will represent the enemy pack after several have been destroyed and the player has just destroyed the current left most enemy.

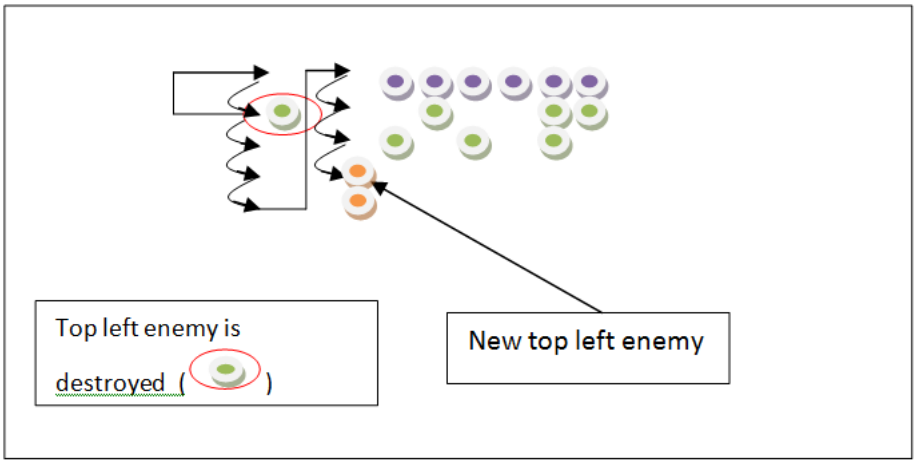


Figure 35 – Finding next left most enemy in remaining pack

The proposed algorithm to find the next left most enemy (as portrayed in the above image) is as follows:

If left most enemy is destroyed then

Go to the first enemy number in the column

While new left most enemy is not found then

Check all enemies in the column and see if they are active

If no enemy is found then move to the next row

Else

New left most enemy has been found

This algorithm is then replicated and altered to find the right most enemy and this should be a sufficient solution.

Creating and Updating a High Scores File

The main objective in the game is to achieve the highest possible score and declare yourself as the "best" at the game. As shown in Figure 10, a very basic layout for the high scores screen is to be used in the game. The next step is to plan how the use of high scores will be achievable and to make this aim realisable.

In order to save the high scores achieved, the game will have to create a file and store it in internal memory. Therefore a suitable destination, file type and data to be written to the file must be considered. The plan is to store the high scores file in the public directory in the C:\ Drive as this does not require administrative rights to access/create/modify files with. A bubble sorting algorithm will then be used to sort the high scores for which the algorithm is below:

For each element in the high scores file

Compare player score to high score

If player score >= high score Then

Temp = high score[i]

High score[i] = player score

Player score = temp

The algorithm ensures that any player that achieves a score equal to that of another player then their score takes preference as it is more recent. This will encourage previous players to continue playing the game in order to check their high score is still intact.

Implementation

Challenges

Learning C# and XNA Game Studio

One of the major challenges of this project was to become affluent using C#, as well as becoming familiar with the Visual Studio environment and XNA Game Studio framework. However there are plenty of online resources and downloadable tutorials, of which one of the most important was Game Development Tutorial (Microsoft).This provided an invaluable insight into how C#, Visual Studio and XNA Game Studio worked and allowed for basic implementation to begin shortly thereafter.

Unfortunately however more in depth research was required in order to gain an understanding on how to do "simple tasks" in C# but more specifically understanding how each component of the XNA Game studio game engine functioned. Thankfully a thorough educational file Learn Programming with XNA by Miles, R. Sithers, Andrew. (2009) was available from App Hub (Microsoft), provided a step by step guide through each component.

Xbox 360 Testing

An unforeseen problem with regards to creating an Xbox 360 version of the game was that in order to test the program on the console a developer fee was required to do so. Therefore the decision was made to make the PC version of the game compatible with the Xbox 360 controller in order to illustrate how the program would be played with an Xbox 360 console.

Positioning Accuracy

Early in the project development a slight issue arose with regards to the scaling of each graphic to be drawn to the screen. Unfortunately there is a drawback of using scaled textures within bounding boxes, which is the loss of accuracy of the player's position on the screen due to the restrictions of using a rectangle in the game.

In Layman's terms, without a bounding box around the player graphic, the position of the player is represented as a decimal value. However when using a rectangle to encase the graphic, (the rectangle position acts as the player's position) and due to restrictions in the Rectangle class, the coordinate values must be integers. For example if the players position on screen is:

X coordinate: 52.5 and Y coordinate: 98.449

Then the players equivalent position with the bounding box applied would be:

X coordinate: 53 and Y coordinate: 99. This therefore creates a minor loss in screen position. However as this is such a minute difference in position, the effects will be the most noticeable when updating the players movement.

For example a player that moves across the screen 0.1 of a pixel at a time will give the illusion that it is moving smoothly across a screen. Whereas if the player were to be moving at 1 pixel at a time, it would look like it is jumping across the screen. Thankfully this has not proved to be a major issue and the updating of objects across the screen does not look "jumpy".

Updating Firing Enemies

This was one of the most challenging and time consuming tasks in the project implementation. The original approach (using pointers), proved to be overly complex when removing an enemy from the list once destroyed. The problem was that, it took a lot of effort for the program to figure out where in a list it should update the pointers in order to maintain those that had line of sight of the player. Unfortunately after multiple attempts of implementing this approach the algorithm created only worked for 80% of cases.

Therefore a reluctant decision was made to opt for a much simpler solution with the consequence of having to re-write almost the entire game engine. The new solution focussed on **not** removing an enemy from the list upon its destruction, which allowed for a far simpler algorithm to be implemented updating the enemy pointers. The method for this can be found in Appendix D.

Implementation of Bezier Curve

Despite having done sufficient research and design of how to implement a curved path for the enemy, this proved difficult to implement for a number of enemies and how to manage them. The approach of using a list of lists was attempted to create a list of curved paths for each enemy that was “breaking away” could follow. However the loop had to break after each single iteration (due to speed of loop execution it had to break to ensure gradual movement across the curve), and unfortunately due to time constraints result was unachievable. Therefore the solution had to resort to allowing only one enemy in a specific row being eligible to break away and for one enemy to break away at any time.

Bayesian Network

The implementation of the Bayesian network unfortunately took an overly long time as I struggled to fully understand and appreciate how the formula works. After several attempts the probabilities being produced were exactly the same as those defined in the code and so gave the impression it was just returning the values that it was parsed. Thankfully after lengthy discussions with Dr. Langbein a working solution was implemented.

There was also the intention for this project, to implement a Bayesian network for the enemies to decide whether or not to break away. However due to prior problems with implementing the Bezier curve and time constraints, this feature was not implemented.

Accessing Internal Storage Device for High Scores

Unfortunately due to being relatively new to using C# and XNA Game Studio it took quite some time in order to find a suitable approach to creating and updating a high score file for just the Windows OS PC version of the game. There were many different approaches of implementing a high score file available online, but most were out dated and were not compatible with the version of XNA Game Studio being used. Therefore a simpler approach was used which was to create a text file which is stored in a public directory of the users “C:\” drive (due to admin rights restrictions). The method for updating the high scores file can be found in Appendix E.

Therefore unfortunately due to time constraints there was no possibility to fully research and implement a solution to create a file on the Windows Phone Version of the game.

Program Robustness

Due to the size and complexity of the overall solution, it required a long a rigorous process to ensure that the program in its "final state" was as bug free as physically possible. This required many hours of testing and getting beta testers to play through the program and report back to me with any crashes or unexpected errors that occurred.

Code Explained

This section of the project will discuss some of the more interesting sections of source code from my project solution and reflect on their impact on the overall solution.

Update Firing Enemy Pointers

As previously discussed, this section of code was a significant challenge for me to overcome. However I am very pleased with the resulting method due to its simplicity and ease to manage in the program which can be found in Appendix D.

The method (as stated in the design section) uses what is called a "jagged array" which is simply a two dimensional array which is used to form a matrix of values. This has been done as the formation of the enemy pack is a matrix and therefore it is easy to visualise each enemy pointer in its matrix form.

Therefore when an enemy is defeated the "UpdateFiringEnemiesList" method is called to do the following:

- Retrieve the enemy position in the matrix
- Determine whether the enemy was either eligible to fire or firing at time of death
- If so then
    - From its position retrieve the value stored there (the pointer to next to fire)
    - Make the defeated enemy pointer equal to null (therefore it cannot be accessed anymore)
    - Make the enemy pointer eligible to fire by changing its state to "has line of sight"
- Otherwise if the enemy was not eligible to fire (this means it is behind an enemy) then
    - Retrieve its pointer
    - Find the enemy in the matrix whose pointer is equal to that of the enemy defeated
    - Make the enemies pointer equal to that of the deceased enemy

Using this simple technique this method ensures that throughout the game, only enemies with an unobstructed view of the player (excluding barriers) will be eligible to fire.

Collision Detection

Thankfully the collision detection was made incredibly easy for me by the XNA Game Studio framework. I made consistent use of the "intersect" method which checks to see if any points in two rectangles intersect. Using this method allowed me to implement collision detection quickly and easily. An example of a collision being detected between the player and an enemy laser would look like so:

```
If(playerRectangle.Intersects(enemyLaserRectangle)){ collision occurred }
```

Bayesian Network

The method for the implementation of the Bayesian network can be found in Appendix C. As mentioned in the interim report and in the design section of this report, the Bayesian network will consist of four nodes; i.e. firing, above player, above barrier, within radius of player. The method makes use of the formula:

P(Firing) . P(Above player|Firing) . P(Above barrier|Firing) . P(Within radius|Firing)

Where P is the probability.

The method then takes the following steps to generate a probability:

- Calculate the first part of the equation by placing in probability values of each node in the network based on whether they are true or false. This value is regarded as the firing being true.
- The method then calculates the opposite equation in order to normalise the result.
- Finally the method normalises the outcome by dividing the values generated.

It is important to note that this method is not computationally expensive, as a probability has to be calculated within a half second in early rounds and within 0.2 of a second in later rounds. Therefore it is important that probabilities can be calculated quickly and efficiently.

Bezier Curve and Generate Break Away Thresholds

One of the most unexpectedly interesting aspects of this solution is the generation of a Bezier curved path for an enemy to follow when it breaks away. As discussed previously there were many approaches to create a reasonable solution. However the Bezier curve appeared to be optimal as it allows for simple tailoring to be done changing the amplitude of the curve produced fitting the programmer's needs. The Bezier curve generation method can be found in Appendix B.

The method I wish to discuss in detail however is the "Generate Thresholds" method in the program which utilises the generation of Bezier curve which can be found in Appendix K. The method works by doing the following:

- Retrieving the positions of the left most and right most enemies in the pack upon initialisation
- Start with the right most enemy and create a Bezier curve (to the right)
- Loop through the points in the curve and do the following:
    - If the current largest value is < value in the curve then
        - If value is >= screen width – 20 then
            - Make current value = value in curve
        - Else
            - Break
    - Increase right most enemy position by 10
    - Back to start
- Then repeat the following but for the left most enemy.

As a result this method returns a Vector value (X,Y) where X is the left most position an enemy can be in for it to be eligible to break away and Y vice versa.

Results and Evaluation

Classic Vs Modern Comparisons

In order to assess the differences and compare both versions of the game, a number of experiments were carried. The experiments are as follows:

- Average score obtained per game,
- Average round number achieved per game,
- Time taken for barriers to be completely destroyed (with no user interaction),
- Enjoyment/game satisfaction survey.

For each test the aim is to collect data from users that consider their gaming ability as:

- Beginner (i.e. New to gaming),
- Intermediate (i.e. has gaming experience),
- Advanced (i.e. has a lot of gaming experience).

For ease of use the tests will be undertaken on the PC version of the game and the mobile version where possible.

Average Score

The purpose of this test is to ascertain how easy/difficult it is for a player to achieve a maximum score during the game. It should be the case that a player can achieve a higher score on the classic version of the game however it may be the case that, due to power ups, the player will have greater sustainability and therefore achieve a higher score also.

For this test I have had the assistance of 3 other people who played each version of the game 3 times as well as myself. The scores attained in each game can be found in Appendix F.

The results have been compiled into a bar graph to greater illustrate the differences in scores between game modes.

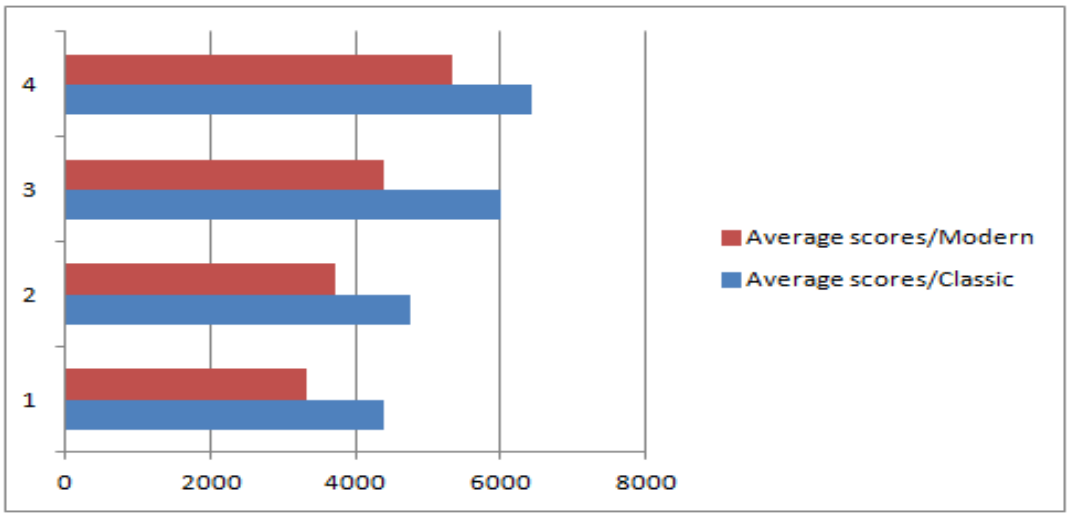


 Figure 36 – Graph to show average score achieved by 4 users in both game modes

From Figure 36 we can see that there is a clear but slight difference in scores with users consistently scoring higher in the classic version of the game and less in the modern game. This will reflect on the average number of rounds each player achieved and will give the impression that the modern version is more difficult.

Average Round Number

This test is very similar to the above in that it will test how many levels a play can survive for before being completely destroyed.

The data gathered from the test (as shown in Appendix G) conforms with the average scores players achieved in both game modes. Again it is evident that players are getting to a higher round number in the classic version of the game as opposed to the modern version.

Again this has been compiled into graphical form to better illustrate these outcomes.

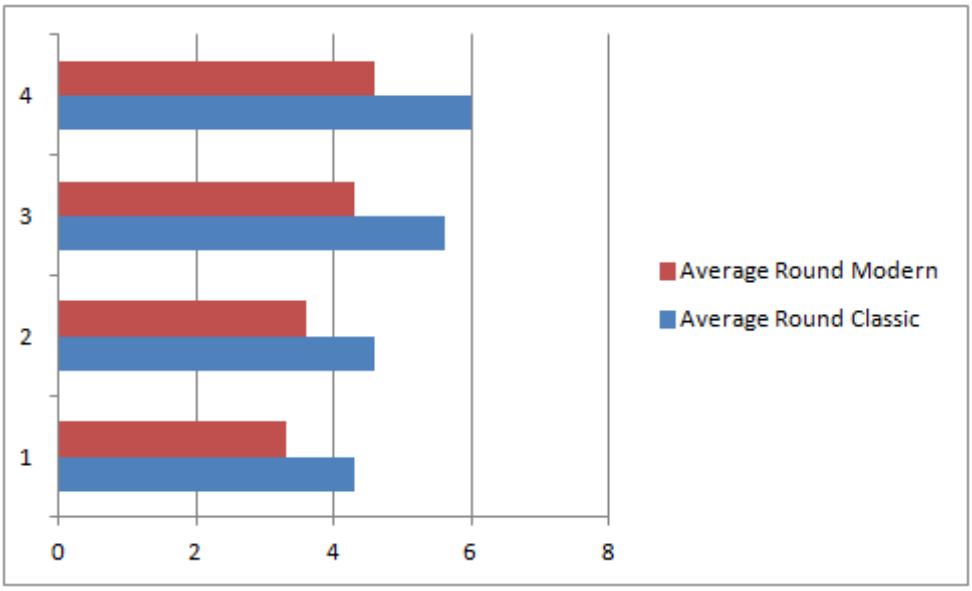


Figure 37 – Graph to illustrate average round number achieved by 4 users in each game mode

Time Taken to Destroy Barriers

This aim of this test is to illustrate how aggressive the enemies are with regards to systematically wearing down the players barriers. This should correlate to the average round number the player reaches. The test will run for the first minute of game play after which point I will assess the damage to each barrier. The results for this test can be found in Appendix H. Again I have used a graph to better illustrate these results to allow for a more effective comparison to be made.

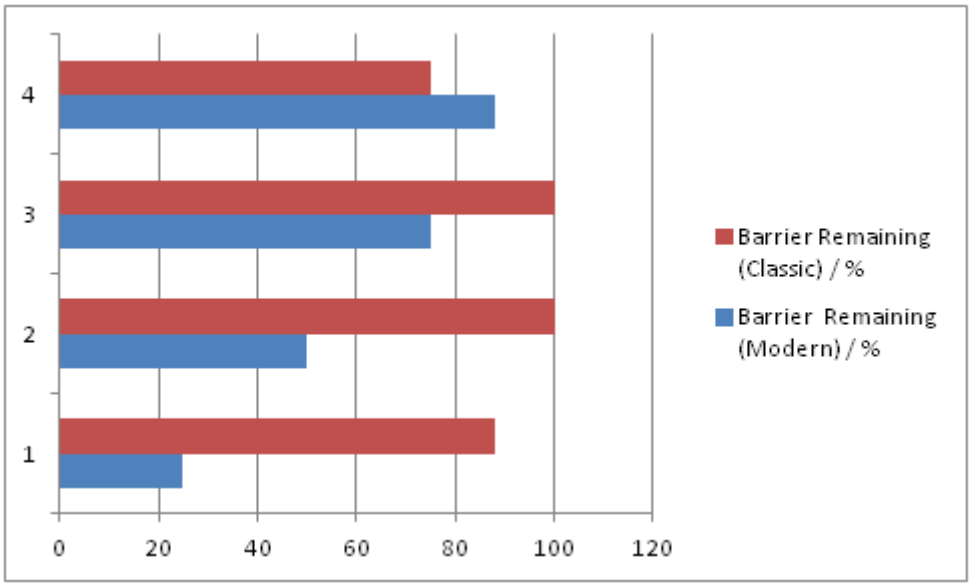


Figure 38 – Percentage of barriers remaining after 1 minute of game play

From Figure 38 we can see that there is a more noticeable difference here between game modes. This proves that in the modern version of the game the enemies are systematically targeting the player's barriers in an effort to expose them and eventually destroy them. It is also noticeable that barrier 1 has been worn down to 20%. Upon further investigation it was found that this is because the player was situated beneath barrier 1 and therefore the enemies were targeting that barrier as a priority.

Enjoyment Satisfaction Survey

This test is as crucial as all of the above as there is no point in the game if the player does not enjoy it. It should also highlight any correlations with regards to the difficulty of the game and the player's enjoyment. This test will ask the player of their thoughts with regards to each version of the game and to note at least one pro and con for each. It will then ask the user to rate the overall game on a scale of 1-10, (1 being, will never play again and 10 being the best game ever and will recommend to all my friends). The results of this survey can be found in Appendix I and of which have been compiled into graphical form to highlight the overall outcome of the survey.

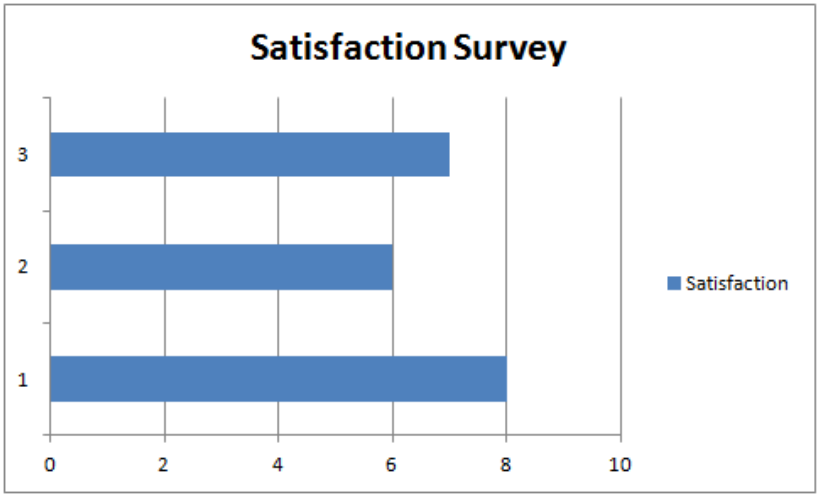


Figure 39 – User Satisfaction Survey

From Figure 39 we can see that the overall feedback from the testers has been positive (no feedback being below 6). The constructive criticism from the advanced and beginner user were conflicted, with one stating the modern game was too easy and the other stating it was too hard. Therefore a solution to this would be to add a difficulty setting to the modern version of the game.

Evaluation of User Interface Design

During the implementation of this game I stuck strongly to my user interface designs shown in the design section of the report. Upon completion I was very happy with them and made very minor alterations to them. In game screen shots can be found below:

Figure 40 – Screenshot of expanded main menu

Figure 40 illustrates the main menu when the user has selected the play option. As you can see the game offers the player two game modes to choose from.

Figure 41 illustrates a basic implementation of a paused screen interface. As you can see, there is nothing flamboyant happening here, the game is simply paused and the user is able to select continue or exit.

Figure 41 – Screenshot of paused screen

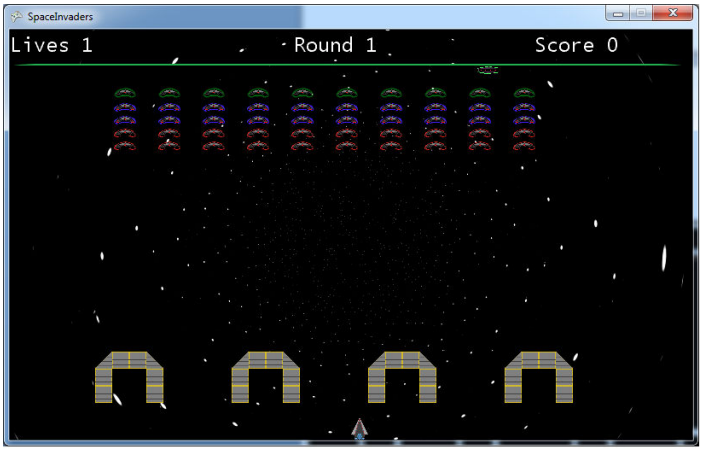


Figure 42 illustrates how the game screen user interface looks. As you can see it is almost identical to that of the design and I am very happy with the outcome.

Figure 42 – Screenshot of play screen

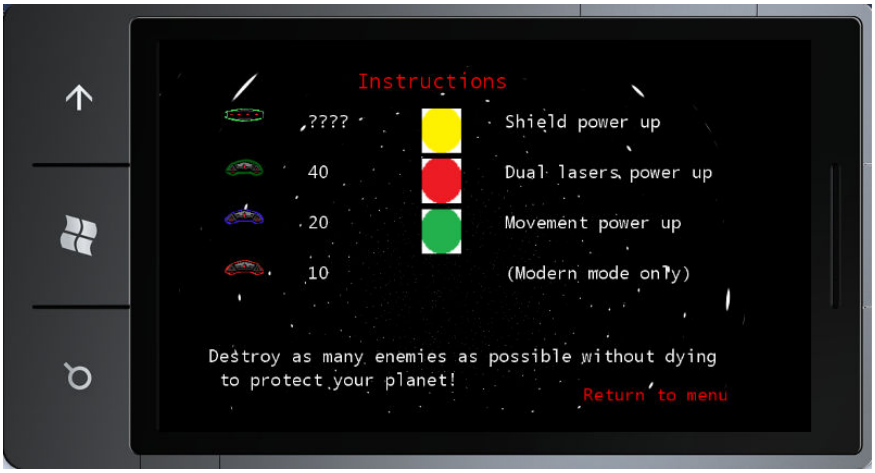


Figure 43 – Screenshot of the instructions screen on Windows Phone emulator

Figure 43 illustrates what the instruction screen looks like on the Windows Phone platform. This screenshot further illustrates how the program is able to scale down all graphics and fonts to fit any screen size and not lose its layout.

Evaluation of Solution against Initial Specifications and Project Aims

The objectives for this project, as noted in the interim and found in Appendix J have been almost wholly satisfied by the solution produced. The following objectives however have **not** been satisfied:

- Tombstone effectively for Windows Phone (basic requirement)
- Include animations (basic requirement)
- Enemy Bosses (optional requirement)
- Multiplayer options (optional requirement)

Unfortunately all of the above have not been satisfied due to time constraints, however I do believe that these objectives carry little weight with regards to increasing value of the overall project solution and therefore I am pleased that only a few objectives were not satisfied by the project.

With regards to the objectives that were satisfied, from the tests carried out during the evaluation phase I believe I have successfully proven that I have implemented a clone of the original Space Invaders game. From the user feedback it has been shown that users can appreciate playing a classic game on a modern platform and that this does not hinder the user's enjoyment. Also my user testing has proven that the enemies in the modern game are far more "aggressive" in that they systematically aim to whittle the player down by shooting at barriers and making every shot count. The feedback has shown mixed results with regards to the enjoyment of this and the average score/rounds achieved may illustrate that the enemies are overly aggressive in the early rounds.

A solution to this would be to decrease the probabilities found in the Bayesian network. However I expected that this would require extensive user testing to find the right balance of probabilities to ensure the game is challenging enough to keep the user engaged in the game but not overly challenging that users get frustrated and would therefore provide a suitable basis for further work on the project.

One final objective I was unable to successfully implement was the use of high scores for the Windows Phone version of the game. This is because the approach taken by the PC compatible version was incompatible due to the way in which the program accessed internal memory. As stated previously, due to time constraints I was unable to fully investigate and implement a solution for this objective.

Evaluation of Project Approach

Due to the major challenge of learning how to use a new programming langue, IDE and framework, I have had to rely heavily on an effective approach to allow for this project to be a success. I have had to make extensive use of an agile development approach as to allow me to design an aspect of the solution, learn how to program it and then physically program it and test it.

Despite the challenges faced, I fully stand by my decision to program the solution using C# and XNA Game Studio. Once I had learned the basics, the XNA Framework was able to provide me with not only a basic game engine architecture but also several game specific methods such as the “intersect” function for collision detection. Visual Studio was of a great help to the implementation of the project as it allowed for me to physically view each and every class along with every texture in one program. It also provided me with accessibility to a Windows Phone emulator of which I could use to extensively test my solution for the Windows Phone.

Future Work

3 Dimensional Game

This game could be made more modernistic by adding 3 Dimensional game play to it. From the solution created it will act as a strong framework to quickly and efficiently create a 3 Dimensional version of the game.

Improved Collision Detection

The collision detection algorithm I am using at the moment is very efficient however it is not very accurate. Therefore in order to provide accurate and satisfying game play it would be worth investigating and implementing a pixel by pixel collision detection algorithm to rectify this.

Collaborative Enemies

One of the advanced objectives of the game was to have the enemies “collaborate” with one another and act as a whole intelligent unit. This would be done by allocating specific roles such as; leader and follower. The leaders would then command the following enemies to do what they will. An interesting idea for further developing this would be to make the following enemies less obedient when there are few remaining. Unfortunately due to time constraints this objective was not satisfied.

Different Break Away Paths

The current project solution provides only two curved paths for an enemy to follow (one to the bottom left hand corner and the other to the bottom right hand corner). As a result this pattern is quickly learned by the player and they can take evasive action immediately to

avoid being destroyed. Therefore by adding a number of different possible paths the player will have to remain vigilant and act at the last moment to avoid being hit.

Difficulty Settings

As mentioned previously, it would be beneficial to implement two difficulty settings for the modern version of the game in order to give advanced/beginner players the best game play experience possible whilst challenging them enough for them to find it competitive.

Conclusion

In summary my project has been an overall success, I have successfully implemented a Space Invaders clone and have made it compatible with Windows OS PC and Windows OS Phone. I have also created a "modern" version of the game that makes use of a Bayesian network which makes the enemies make more tactical shots during the game which makes for a more challenging gaming experience. I have also allowed for certain enemies to break away from the main pack and attack the player head on to add for a further change in game play from the original. From testing I have proved that the modern approach is more challenging but has been found to be either too hard or too easy.

Unfortunately I was unable to implement some of the advanced features proposed for this game which is disappointing as there were some interesting ideas that were left unexplored.

Finally the resulting game is compatible with only the PC and Windows Phone as opposed to the Xbox 360. However the PC version does accommodate for use of an Xbox 360 controller to provide an insight as to how the game would run on the console.

Reflections

This project has provided me with an opportunity to select a project of my choosing and in doing so allowed me to set myself a challenge of which I knew I would find interesting enough to motivate myself to complete it. Initially I thought this project was overly ambitious especially considering I had no experience using C#, Visual Studio or XNA Game Studio however I have learned that over a reasonably short space of time and with a lot of time practice and determination, I have become comfortable and confident programming in C#. However as a consequence I have learned to be cautious when assigning project goals. As I have mentioned throughout the project lifecycle that this project is not only grand in scale on its own, but the challenges involved when learning a new language will almost certainly double, if not triple the amount of time spent on simple coding practices.

With regards to my project reports I have learned that despite the amount of effort and hard work that is put into projects, it is important not to try and include absolutely everything that has been done to the reader. The ability to ensure a report is concise and easy to read for the reader is of upmost importance and unfortunately I have a tendency to waffle due my wanting to portray how much work I have done into report form.

References

Miles, R. Sithers, Andrew. (2009) Learn Programming with XNA. Available at: https://www.facultyresourcecenter.com/curriculum/pfv.aspx?ID=7992&Login=&wa=wsignin1.0&c1=en-us&c2=0 (Accessed at: 28 July 2011)

App Hub. Available at: http://create.msdn.com/en-US (Accessed: 8 August 2011)

App Hub Game Development Tutorial. Available at: http://create.msdn.com/en-US/education/tutorial/2dgame/getting_started (Accessed: 8 August 2011)