MAJOR PROGRAMMING ASSIGNMENT

Stage One and Two

Jack Carey, IS1201

Table of Contents

[Design Specifications 3](#_Toc531763859)

[User’s Perspective 3](#_Toc531763860)

[Developer’s Perspective 3](#_Toc531763861)

[User Section 4](#_Toc531763862)

[Overview of the Program 4](#_Toc531763863)

[Screen Designs 5](#_Toc531763864)

[Description of screen designs from a user’s perspective 6](#_Toc531763865)

[Developer Section 7](#_Toc531763866)

[Level 1 DFD Diagram 7](#_Toc531763867)

[DFD Diagram Description 8](#_Toc531763868)

[Data structures 9](#_Toc531763869)

[Classes 9](#_Toc531763870)

[Arrays 10](#_Toc531763871)

[Records 11](#_Toc531763872)

[Files 11](#_Toc531763873)

[Structure Chart 12](#_Toc531763874)

[Description and purpose of each module 13](#_Toc531763875)

[Function 1: sortHighScores 15](#_Toc531763876)

[IPO Diagram 15](#_Toc531763877)

[Data Dictionary 15](#_Toc531763878)

[Pseudocode 16](#_Toc531763879)

[Function 2: validateName 17](#_Toc531763880)

[IPO Diagram 17](#_Toc531763881)

[Data Dictionary 17](#_Toc531763882)

[Pseudocode 17](#_Toc531763883)

[Subroutine: updatePosition 18](#_Toc531763884)

[IPO Diagram 18](#_Toc531763885)

[Data Dictionary 18](#_Toc531763886)

[Pseudocode 18](#_Toc531763887)

# Design Specifications

## User’s Perspective

* The game should be played with the WASD or arrow keys and a fire button, power up button, and a start/pause button.
* The game should be played with one player, with a possible implementation of two players.
* The game should ask for the player’s name before the game starts.
* During gameplay, the game should track and display the player’s name, score, number of lives, and the high score, along with the “power meter”, a bar which shows the abilities and upgrades available to the player.
* The player should control the space fighter called the “Vic Viper”. The player controls its movement, and can fire the Vic Viper’s weapons.
* The player should be able to unlock a variety of different power ups which affect the Vic Viper’s weapons and movement.
* The player should be confronted with a variety of enemies from the right-hand side of the screen, which will destroy the Vic Viper on impact or on enemy projectile impact.
* The game should include pixelated, arcade-style graphics, and include sound effects and music.
* The screen and background should scroll to the left to give the illusion of movement to the right.
* When the Vic Viper is destroyed, the player should lose a life.
* When the player runs out of lives, the game should end and the high scores displayed.

## Developer’s Perspective

* The player’s name and score should be stored throughout the entire game.
* The player’s name should be validated to ensure it complies with the requirements for the name.
* The player’s scores should be stored in a sequential file, which will be sorted to ensure the highest score will be on top for displaying.
* There should be an internal way of representing a level, or the physical features of the game map, including the terrain geometry and hitboxes, textures, and the spawning locations and times of enemies.
* Entities should inherit from a single entity object which will expand from there, including the player, options (a certain power up), enemies, projectiles, and power-ups.
* Entities should use a custom WPF control to ensure they can control themselves easily.
* A game loop needs to be implemented.
* The positions and hitboxes of all entities should be tracked to ensure collisions can be calculated.
* Entities should have properties including movement speed, position, texture array, hitbox bounds, projectile type, and movement rules.
* Entities should be managed correctly - they should be destroyed and cleaned from memory when they are destroyed or leave the screen.
* Code should be planned and organised well in advance - this is a complicated project with many processing paths and care should be taken to minimise clutter and “spaghetti code”. This can be accomplished using multiple files and modules, along with careful use of classes and objects.

# User Section

## Overview of the Program

Gradius is a single player game where the player controls the spacecraft known as the “Vic Viper”. They must manoeuver the Vic Viper to avoid terrain, enemies, and enemy projectiles. The game is a side scrolling space shooter; as the screen continuously moves itself to the right, new obstacles and enemies are revealed. The game uses a unique power-up system called the “power meter” which players must utilise to purchase different types of power-ups. When the Vic Viper collects a power-up item, a specific power up is selected on the power meter, and the player can either choose to purchase that power-up or wait and save up for a more powerful power-up. The game is split into different stages with different terrain types, however for this recreation only the first stage will be playable.

The Vic Viper starts the game with a slow movement speed, and a single weak gun which fires straight in-front of the space fighter. As different formations of enemy types come from the right side of the screen, power-up items are dropped periodically, so the player can upgrade their ship with different weapons. In order, the power meter includes: Speed-Up, which increases movement speed; Missile, which are air-to-surface missiles, Double, which adds another upwards facing gun; Laser, which replaces bullets for enemy penetrating bullets; Option, which adds an invincible drone which follows the Vic Viper and shoots with it; and “?”, which is a shield for the Vic Viper.

There are a variety of enemy types which follow certain movement patterns depending on the player’s position. As the Vic Viper will be destroyed immediately on contact with an enemy, enemy movement patterns should be understood to avoid losing a life. Enemies also shoot projectiles towards the player, which will also destroy the ship on contact. The Vic Viper should aim to destroy as many enemies as possible to gain as many points and avoid being destroyed.

The aim of the game is to earn as many points as possible, and make it to the end past the boss known as “Big Core”. The high scores of players will be saved, with a possible online implementation.

## Screen Designs

## Description of screen designs from a user’s perspective

The above screen designs give an overview of the entire game, and the path that users will take throughout the application. It is a rough outline of how each screen will look in the final app, the layout of controls and text, and the colour scheme of certain UI elements.

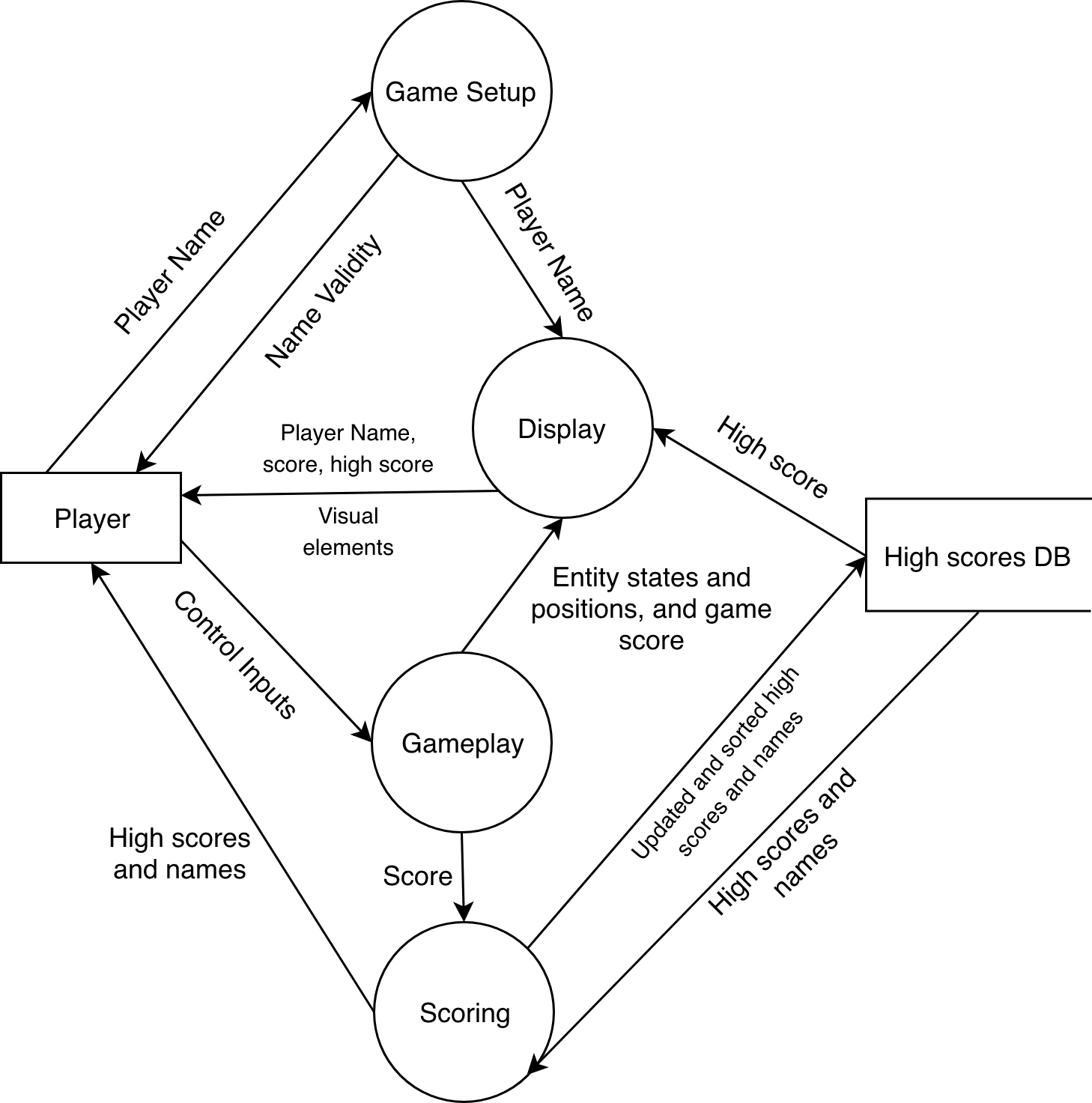
The first screen, at the top left, is the first screen presented to users, and includes a way for the user to enter their name, and start the game quickly. It also contains a help button, which users can press for assistance playing the game. This means that the first screen is both efficient for existing users and accessible for new users. There is also a button to toggle the sound of the game on or off.

The second screen is the main screen where users will spend the most of their time. The “GAMEPLAY” box is where the Vic Viper, terrain, enemies, and other gameplay elements will be. The elements along the bottom of the screen include the player score, name, lives, and the high score. These text elements will be updated to display the respective data. The power meter is the blue boxes below the GAMEPLAY box, and was described in detail in the “Overview of the program”.

The third and final screen is the screen users will see when they run out of lives in the game. If the player achieved the highest score, the message “You got the high score!” will appear. A list of all the high scores will then be displayed in the middle of the screen, including initials and the respective score. The option to play again, and return to the splash screen is presented, however at this point the user can simply exit the app using the operating system supplied controls at the top right of the window.

# Developer Section

## Level 1 DFD Diagram



### DFD Diagram Description

The above DFD outlines the flow of data between different sections of processing in the game. The game has been split up into 4 distinct processes: Setup, Gameplay, Display, and Scoring.

Setup receives the player name from the player, ensures it is a valid name, and prepares the application for gameplay.

Once the game is ready to start, the Display and Gameplay functions work in tandem to provide the gameplay to the player. The Gameplay process manages player input, including the movement inputs for the Vic Viper, and the control to fire the space fighter’s weapons. It then processes this input with all the different gameplay elements, determines if there were any collisions, updates the positions of all on screen entities to suit, and sends this information to the Display process. This process executes the appropriate functionality to display the gameplay elements to the user, including graphics, the positions of entities including the player, enemies, power-up items, projectiles, the current score, and the high score, which is fetched from a high score file.

Once the game is finished - when all lives have been depleted - the application moves to the scoring process, which fetches all the high scores from a high score file, sorts them, and displays them with the current game score to the player next to the respective player names. It then sends this updated list of high scores back to the high score file, which will now contain the new score generated from the current game as well.

## Data structures

### Classes

Classes are like records (structures) in the way that they provide the blueprints for an object. They give the properties that the object has, and unlike records also provide methods that the object can use. They can inherit from other classes and use their properties as well, and allows the developer to expand on and create variations of a single thing. For example, a class can be defined and called Car, and other classes can be defined called Hilux, i30, or Civic which inherit the same common attributes from Car but also add their own, like trayVolume or bootVolume, which isn’t applicable to all Cars.

**CLASS NAME:** Entity

**INHERITS FROM:** None

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Property or Method | Signature or Data Type | Description |
| position | Property | Point (built-in structure) | Represents the entity’s location in game (from the top left of the entity) |
| hitBox | Property | Size (built-in structure) | Represents the size of an entity’s hitbox (generated from the size of the WPF control). May be more complex (possibly a method) for triangular shapes/rotations. |
| control | Property | UserControl | A reference to the WPF control the entity manages. |
| movementSpeed | Property | Integer | A multiplier to determine how fast an entity moves |
| direction | Property | Vector (built-in structure) | A vector to denote the next direction of movement the entity will travel in. This will be multiplied by *movementSpeed* to give the correct displacement. |
| move | Method (Subroutine) | move() | Moves the entity in the direction of *direction*, while also considering the constant leftward movement (vector subtraction). |
| destroy | Method (Subroutine) | destroy(ByVal onScreen As Boolean) | Play the explosion animation and destroy the object, and remove it from memory. |

**CLASS NAME:** VicViper

**INHERITS FROM:** Entity

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Property or Method | Signature or Data Type | Description |
| projectileSpeeds | Property | Tuple | Represents the speed of the Vic Viper’s projectiles. The Tuple is 2 elements long. The first element is for the main weapon (gun or laser) and the second is for missiles. |
| powerUps | Property | Record (PowerUps: see below) | Represents the current power ups the Vic Viper has. |
| lives | Property | Integer | Represents the amount of lives the Vic Viper has left. |
| projectileCount | Property | Tuple | Counts the number of projectiles the Vic Viper has fired and is currently on screen. |

### Arrays

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Array Name | Dimensions | Indexing | Data Type | Description | Scope |
| activeEntities | 1 | 0 to n: at least one element | Entity | Contains all the entities which currently on the screen. The first element will always be the Vic Viper. | Global |
| enemyFormation1 | 1 | 0 to 3 | Entity | Contains 4 different enemies in a group | Local |

### Records

**RECORD NAME:** PowerUps

|  |  |  |
| --- | --- | --- |
| Field Name | Data Type | Description |
| speedUp | Integer | Represents the number of speed power ups the player has purchased. |
| missile | Boolean | True if Missiles have been purchased, False otherwise. |
| gunType | MainGunType (Enumeration) | “Double” if Double has been purchased, “Laser” if Laser has been purchased, and “Normal” otherwise. |
| optional | Integer | Represents the number of optionals purchased: Max of 2 |
| shield | Integer | Represents the amount of hits a shield can absorb before it collapses. Is set to 5 upon purchase, and decremented on every hit. |

**RECORD NAME:** HighScore

|  |  |  |
| --- | --- | --- |
| Field Name | Data Type | Description |
| name | String | Represents the name of the player which achieved the given score. |
| score | Integer | The score achieved by a specific player. |

### Files

One local file will be used for the high scores, which may or may not be updated with an online service.

**FILENAME:** GradiusVB\_Scores.txt

**FILE LOCATION:** Application.UserAppDataPath (Which equates to the appropriately named AppData folder)

**FILE TYPE:** Sequential

**FILE STRUCTURE**: one score and name pair per line, in the format of “name:score”. No sentinel Value. My file will store a maximum of 100 scores at a time - after too many scores have been generated, the worst scores are dropped from the file to ensure the file does not get too large.

**SAMPLE:**

JSC:9999999

PYD:4956288

GJD:304455

NOB:3500

...

## Structure Chart

## Description and purpose of each module

|  |  |  |
| --- | --- | --- |
| Subroutine/function | Control Structures | Description of purpose |
| setup | * Sequence * Pre-test repetition | Manage name collection and set the game up for gameplay. |
| validateName | * Binary selection | Function to confirm the player name meets certain requirements. |
| setupLevel | * Sequence | Set up game objects including the level and the Vic Viper. |
| instantiatePlayer | * Sequence | Initialise the player object (Vic Viper object) with a starting position and default states. |
| loadLevel | * Sequence | Loads the level (including terrain, enemy spawn locations, terrain features) and readies it for gameplay. |
| startGame | * Sequence | Closes the first form, opens the form for gameplay, and starts the game loop. |
| gameplay | * Sequence | Umbrella function to start the input handler and the game loop. |
| inputHandler | * Sequence | Start and set up a handler to manage user input throughout the game loop. |
| gameLoop | * Sequence | Note that this subroutine does not include a loop, but is instead repeatedly called by an external Timer while the game is running.  Main game processing. Is likely to be run once per frame. |
| scrollMap | * Sequence | Move the terrain, including background, to the left. Activate new entities by adding them to the visible entities array. |
| processEnemies | * Sequence * Binary Selection * Multi-way selection | The AI for enemies: Checks the enemies’ location and collisions and updates the specific enemy’s next direction. |
| updateEntities | * Sequence | Updates the states of all entities. |
| updatePosition | * Sequence | Updates the position of all entities. |
| shoot | * Sequence * Binary Selection | Shoots for the specified entity. For the player, only shoots if the button has been pressed and there aren’t more than two bullets currently on screen. For enemies, shoots based on their position and a random timing. |
| checkCollisions | * Sequence | Checks collisions for the entities which need to be checked. |
| checkPlayer | * Binary selection | Checks if a bullet or another enemy has collided with the player. |
| destroyPlayer | * Sequence | Destroys the player, and resets the game after deducting a life. |
| checkProjectiles | * Binary selection * Counted repetition | Checks if player projectiles have collided with an enemy or gone off the screen, if so, destroys the projectile. |
| destroyEnemy | * Sequence | Destroys the given enemy. |
| render | * Sequence | Calls the display module subroutines. |
| display | * Sequence | Updates the entire display and gameplay elements. This may be integrated into WPF control functions. |
| updateTerrain | * Sequence | Moves the entire terrain elements to the left at a certain speed, including the background. |
| updateEntities | * Sequence * Counted Repetition | Updates the positions and the animation frames of all visible entities. |
| updateUI | * Sequence | Updates the score, high score, and lives when applicable. |
| scoring | * Sequence | Manages the displaying of high scores once the player’s lives reach zero and the game ends. |
| fetchHighScores | * Sequence * Counted Repetition | Fetches the current high scores from a file and returns them as an array of records. |
| sortHighScores | * Sequence * Counted Repetition * Binary Selection | Sorts the list of high scores as well as the added game score. |
| writeHighScores | * Sequence * Counted Repetition | Writes the given list of sorted high scores to the high score file. |
| displayHighScores | * Sequence * Counted Repetition | Displays the high scores on screen. |

## Function 1: sortHighScores

### IPO Diagram

|  |  |  |
| --- | --- | --- |
| Input | Process | Output |
| High score array | While array is not sorted;  Let n be the length of the array minus one.  Loop through every element of index i in the array to index n;  If the score field of element i is less than element i + 1, swap element i and i + 1  If no changes are made through entire array, array is sorted. Otherwise, decrement n. | Descending sorted high score array |

### Data Dictionary

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Variable Name | Data Type | Length | Description | Scope |
| highScores | Array of HighScores (Records) | Max 100 | All the high scores and names. | Local (parameter) |
| last | Integer | N/A | The last index the sorting algorithm should sort to. | Local |
| swapped | Boolean | N/A | A flag to check if the sorting algorithm has made an entire pass without swapping anything, which means the array is completely sorted. | Local |

### Pseudocode

BEGIN sortHighScores(highScores)

last = length of highScores - 1

swapped = True

WHILE swapped = True

swapped = False

FOR i = 0 TO last - 1

IF highScores(i).score < highScores(i+1).score THEN

temp = highScores(i)

highScores(i) = highScores(i+1)

highScores(i+1) = temp

swapped = True

END IF

NEXT i

last = last - 1

ENDWHILE

END sortHighScores

## Function 2: validateName

### IPO Diagram

|  |  |  |
| --- | --- | --- |
| Input | Process | Output |
| Player name | If player name length is not equal to three, return false.  Loop through each character of player name;  If the character is not an uppercase letter of the alphabet, return false.  If loop runs through (all characters are uppercase and of the alphabet), return true. | Name validity (Boolean) |

### Data Dictionary

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Variable Name | Data Type | Length | Description | Scope |
| playerName | String | N/A | The player’s name passed in | Local (parameter) |
| valid | Boolean | N/A | A flag to check if the name is valid or not - if it is set to false at any time, the name is not valid. It is initialised to True. | Local |

### Pseudocode

BEGIN validateName(playerName)

valid = True

IF length of playerName <> 3 THEN

valid = False

ELSE

FOR i = 0 TO 2

IF playerName(i) is not uppercase alphabetical THEN

valid = False

END IF

NEXT i

END IF

validateName = valid

END validateName

## Subroutine: updatePosition

### IPO Diagram

|  |  |  |
| --- | --- | --- |
| Input | Process | Output |
| Entity | Multiply entity’s *direction* vector by its scalar *movementSpeed*, and assign the new vector to *scaledVector.*  Add the x component of *scaledVector* to the x component of the entity’s *position*, and respectively with the y component. | Position change |

### Data Dictionary

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Variable Name | Data Type | Length | Description | Scope |
| entity | Entity | N/A | The entity to change position. This function is run once for every visible entity. | Local (parameter) |
| scaledVector | Vector (built-in structure) | N/A | An adjusted vector to match the entity’s movement speed. It is the *direction* Vector scalar multiplied by *movementSpeed*. | Local |

### Pseudocode

BEGIN updatePosition(entity)

scaledVector = entity.direction scalar multiplied by entity.movementSpeed

entity.position.x = entity.position.x + scaledVector.x

entity.position.y = entity.position.y + scaledVector.y

END updatePosition