Programming Project Design

By Farhan Mirza

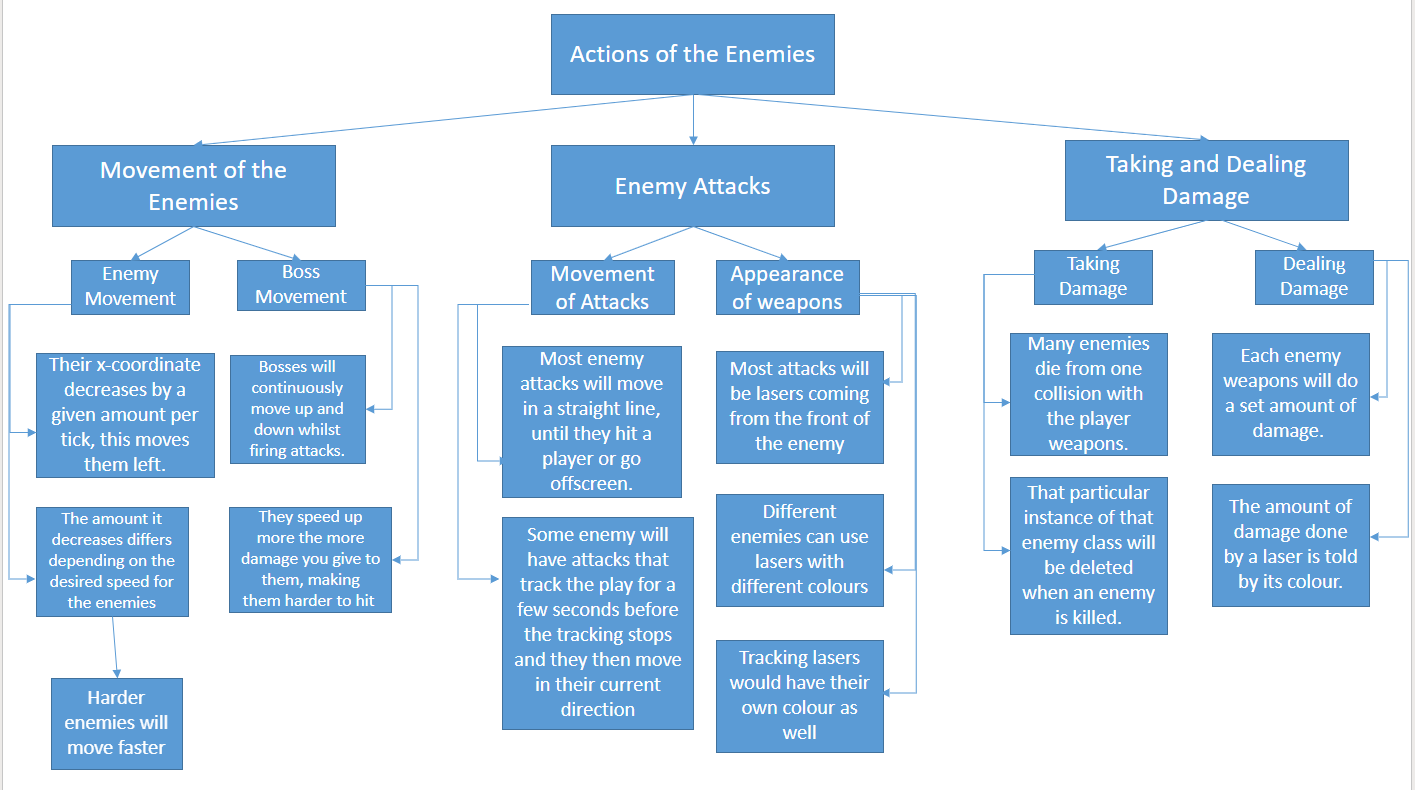
Decomposition of the Problem

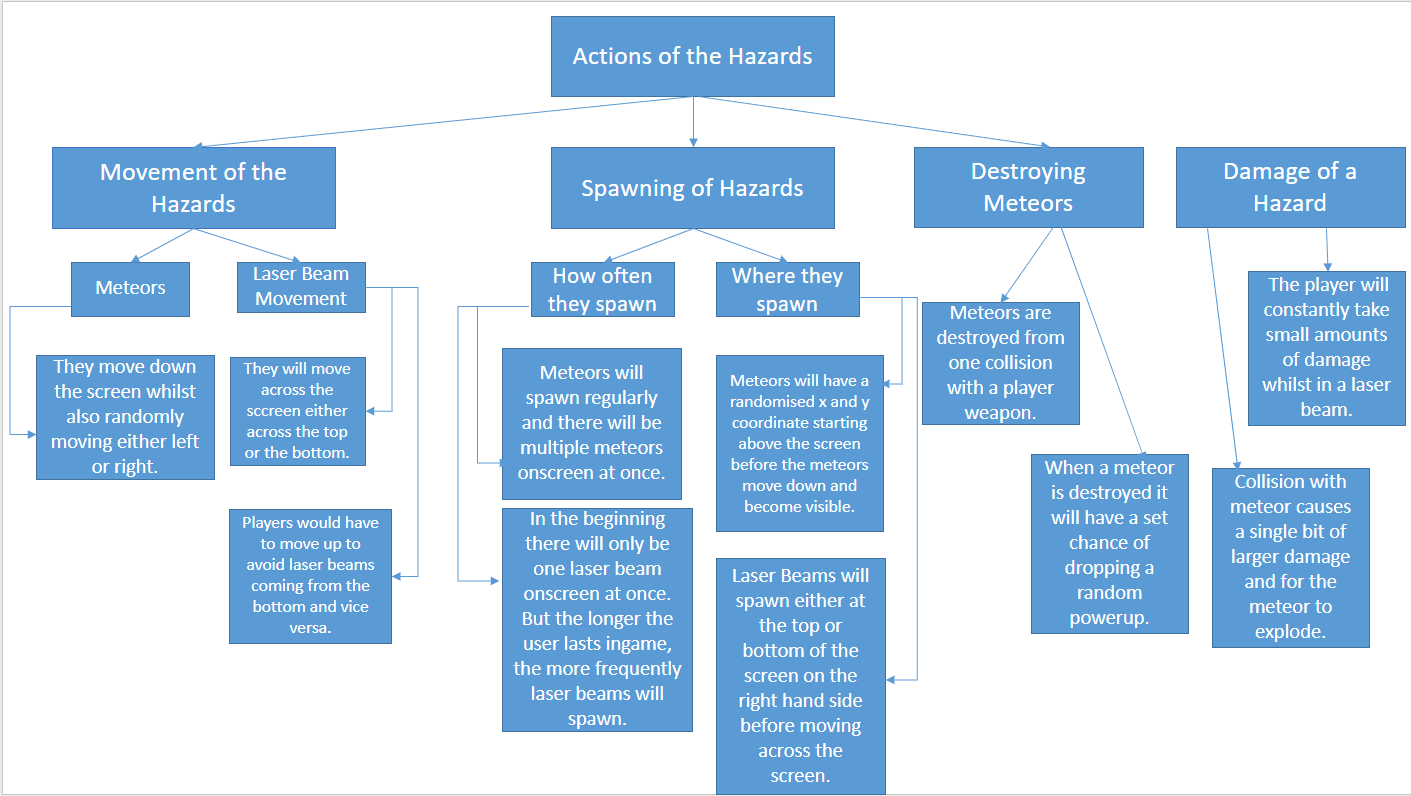
The main problem of a 2D wave-based shooter can be broken down into subsections like:

1. Actions of the Player
2. Actions of the Enemies
3. Actions of the Hazards

These areas can be broken down further using hierarchy charts.







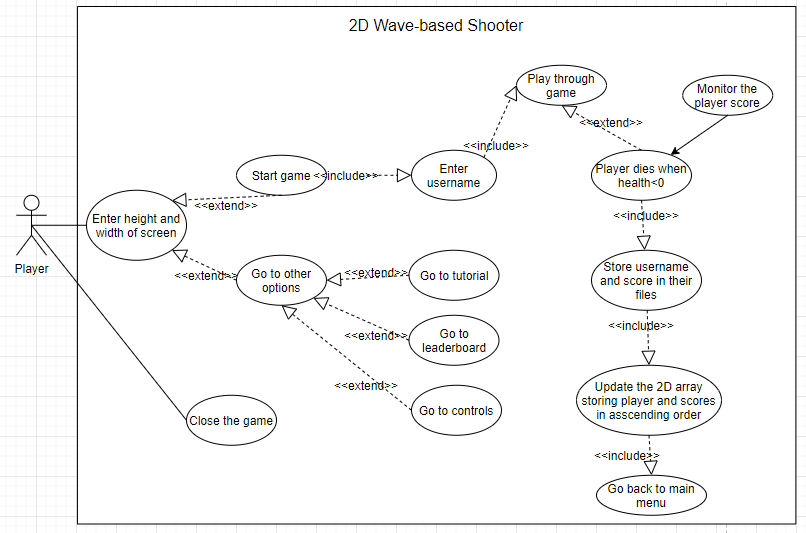
These hierarchy tables break down the three main parts of the game into their fundamental components. For the actions of the player, their movement using the arrow keys was used because it is logical for the player and easy to learn. I am also planning to include a dash feature so that the player has more options on how to dodge enemy attacks which increases how skilled they can be at the game. My design uses the bomb and laser as the player weapons as they both damage the enemy in different ways, with bombs moving vertically and lasers moving horizontally, and this means the player will have to decide which weapons is better to use in each circumstance, which increases how skillfully the game can be played. Also, having player damage that differs depending on the enemy weapon means that enemies can become harder over time as new enemies introduced later will deal more damage to the player. Moreover, having explosion animations tied to weapons hitting player and enemies makes the game more visually appealing and more believable for the player. Also, having some enemy weapons inflict other effects on the player for a short amount of time like slowing them down will make certain enemies much more dangerous than others and they would have to be prioritised in each wave.

For the actions of the enemies, they spawn in on the right-hand side of the screen so making them move leftwards means that they will start off far from the player but will then move closer over time. Making later enemies move faster will also increase the game’s difficulty as the player will have less time to move out of the way from enemies that could fly into them. Bosses only move up and down at the right-hand side of the screen because I don’t want them leaving the screen until the player has beaten them and them moving up and down makes them more difficult to hit. Most enemy lasers will move very similarly to player lasers, but different lasers will have different colours which represent the different speeds and effects of different enemy lasers. For example, orange lasers could represent slow lasers and purple lasers could represent lasers that slow down the player on impact. Tracking lasers will be used with enemies that appear later in the game to increase the game’s difficulty since they will be harder to dodge, especially whilst avoiding the rest of the weapons and hazards. Also, most enemies can be killed in one or two shots because there will be end up by many enemies onscreen at once during later waves so the player will already have a lot to dodge around and by making regular enemies able to take more damage, the game would probably become quite unfair since the player wouldn’t be able to destroy all enemies in a wave before they collided with the player.

For the actions of the hazards, there are two main hazards that were described in the table: meteors and laser beams. The meteors will appear from the top and move downwards to make sure that players won’t just stay at the top of the screen and drop bombs on their enemies from there whilst avoiding most potential damage. The laser beams are also used to keep the player on the move as they move across the width of the screen and when used at the same time as the meteors in later waves the game will become harder as the player will have to dodge vertical and horizontal hazards whilst also fighting enemies. Meteors will spawn in quite frequently and they have a chance of dropping powerups when destroyed. This will be done because it makes the player more alert as they know that any meteor destroy could hold a powerup that could give them health, ammo or other buffs. This encourages some risk-reward gameplay because they will want to stay close to the meteor so they can quickly pick up powerups if they drop, but not to close otherwise the meteor might collide with them.

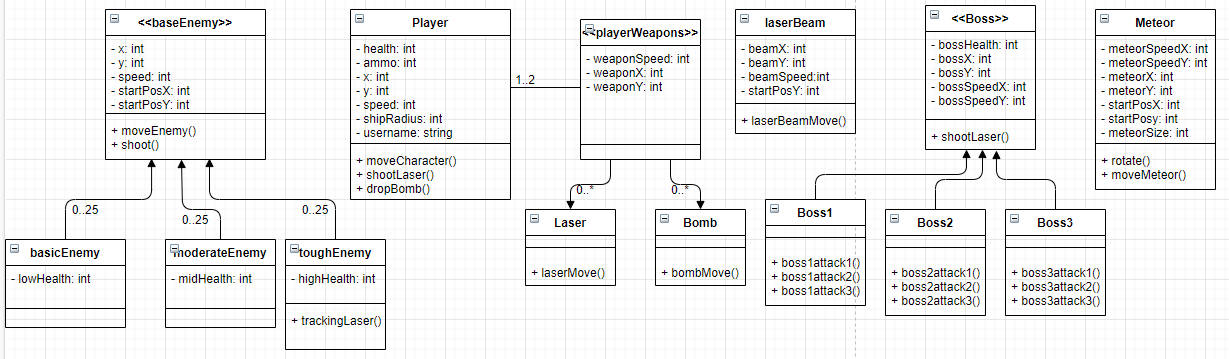
Structure of the Solution

Use Case Diagram for the Game



This use case diagram shows a basic structure of the main features of the program and the order in which they’ll be accessed. The main menu of the game cannot be accessed until the user enters an acceptable height and width for the game screen. The main menu will let the player either start the game, go to a tutorial, go to a leaderboard or view the controls and from each of those sections they will be able to return to the main menu. When the user starts the game, they will have to enter a username which will be stored along with their score at the end of the game. After they enter a username, the game begins, and the player will control a spaceship and fight off wave after wave of wave of enemies by shooting at them using lasers or bombs whilst simultaneously avoiding enemy attacks and general hazards like meteors. The meteors will also sometimes contain powerups like extra health and ammo for the player to use. After a certain number of waves, the player will fight a boss and then move onto the next set of waves until the next bossfight. Whilst the player is fighting enemies, the game will be keeping track of the player score which is increased each time they destroy an enemy spaceship, with harder enemies giving more points. This cycle will continue until the player loses all their health at which point the game will take the username and final score and store them in separate files. The game then takes the scores and usernames and from their files and arranges them in ascending order so that the leaderboard in the main menu is updated. Finally, the program will return the user to the main menu where they can start the game again or do one of the other options on the main menu.

Class Diagram for the Game



This class diagram shows the main classes that will be used in the game. There are a collection of classes, base classes and subclasses in the diagram as many of the objects in the game will share common attributes and methods. All classes will have x and y attributes as all objects in the game will be at a certain position on the screen at a given time. For the enemy classes, they will all move at a given speed and be able to shoot at the player, but they will all have varying amounts of health so the attributes storing the health for different subclasses store different values so that harder enemies take more hits to killed. The toughest enemy also has an additional method which will let them shoot a laser that tracks the player’s location and moves in that direction for a short period of time. The player class shares some attributes with the base enemy class, but also has a ammo attribute for its lasers as well as a username attribute for whatever username the player enters at the beginning of the game. The method it uses to move the player is also different from the one for enemies as enemies will always be moving in a single direction until they get shot or go offscreen, whereas the user will move around the player spaceship. The shootLaser() and dropBomb() methods are used for the players to main attacks and call objects from either the laser or bomb class respectively. The base class for player weapons has attributes for the weapons’ speeds as well as their x and y coordinate which start off as the player’s current coordinates before moving in another direction. The two subclasses have different methods for moving the weapon, as lasers will move horizontally whereas bombs will move vertically. There isn’t a method for dealing with collisions because Pygame can deal with collisions outside of classes when each object is a part of a sprite group and it’s the collisions between different sprites that are monitored instead. The laser beam class similarly doesn’t have a collision-dealing method but instead has method to move the laser beam across the screen at a given speed determined by the beamSpeed attribute. There is also an attribute for the starting y coordinate of the laser beam since it will either appear at the top or the bottom of the screen so that player can’t feel too safe at any given location. The other main hazard, the meteor, includes a start x coordinate for x as well the meteors will always spawn at a random point above the screen. In addition, the meteor has separate speeds along the x and y axis since it will move downwards but will also move to the left or the right as well which requires different values meteorSpeedX attribute. Moreover, the meteor class has a method for moving the meteor downwards but also has a method that will make the meteor rotate around in a circle to make it look more believable. The other base class is the boss class which contains many of the same attributes as the enemy class with a similar purpose, but also has separate x and y speeds as bosses should be more mobile than regular enemies so that they are a tougher opponent for the player. All the bosses will be able to shoot lasers at the players, but they will all also have their own unique attacks which are carried out by methods unique to each boss subclass.

Algorithms

Algorithm 1

**Algorithm letting the player move up, down, left and right and letting them dash (a method in the Player Class):**

PUBLIC Procedure moveCharacter()

INPUT keyPressed

IF keyPressed== “up”:

player.y=player.y+10 pixels

player.moveUp=True

player.moveDown=False

player.moveLeft=False

player.moveRight=False

ENDIF

IF keyPressed== “down”:

player.y=player.y-10 pixels

player.moveUp=False

player.moveDown=True

player.moveLeft=False

player.moveRight=False

ENDIF

IF keyPressed== “left”:

player.x=player.x-10 pixels

player.moveUp=False

player.moveDown=False

player.moveLeft=True

player.moveRight=False

ENDIF

IF keyPressed== “right”:

player.x=player.x+10 pixels

player.moveUp=False

player.moveDown=False

player.moveLeft=False

player.moveRight=True

ENDIF

IF keyPressed== “lshift”:

IF player.moveUp==True:

player.y=player.y+100 pixels

ENDIF

IF player.moveDown==True:

player.y=player.y-100 pixels

ENDIF

IF player.moveLeft==True:

player.x=player.x-100 pixels

ENDIF

IF player.moveRight==True:

player.x=player.x+100 pixels

ENDIF

ENDIF

ENDPROCEDURE

**Trace table for the player method:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **keyPressed** | **player.moveUp** | **p.moveDown** | **p.moveLeft** | **p.moveRight** | **Player.x** | **Player.y** |
| - | False | False | False | False | 500 | 500 |
| up | True | False | False | False | 500 | 510 |
| lshift | True | False | False | False | 500 | 610 |
| left | False | False | True | False | 490 | 610 |
| lshift | False | False | True | False | 390 | 610 |
| down | False | True | False | False | 390 | 600 |
| lshift | False | True | False | False | 390 | 500 |
| right | False | False | False | True | 400 | 500 |
| lshift | False | False | False | True | 500 | 500 |

This algorithm is important as it gives the user the functionality to move in any direction they want as well as letting them dash in the direction they are currently moving in. It uses several Boolean variables to keep track of whether the player is moving up, down, left or right so when the player dashes the direction they dash in will depend on whichever Boolean value is True. The dash makes the player move 100 pixels instead of the 10 pixels for moving regularly which lets a player perform last-minute dodges of enemy weapons and other hazards. Also, the code uses “if” statements only instead of several “else if” statements for the next directions. This is so that if the player is moving up, then they will be able to move to the left or right at the same time which allows them to move diagonally, rather than only being able to move in one direction. This is important as it gives the player more options to move away from enemies and hazards which increases how skilled someone can be at the game. This method in the player class will be one of the most important algorithms in the whole game since moving around to both deal and avoid damage is very important.

Algorithm 2

**Algorithm for taking players and corresponding scores from their files and storing them in a 2D array. It then uses a bubble sort to sort the players and scores by an ascending order of scores:**

players=[]

scores=[]

scoreList=[[]]

myFile=openRead(“usernames.txt”)

FOR x=0 to myFile.endOfFile():

players.append(x)

next x

myFile.close()

MyFile2=openRead(“scoresAchieved.txt”)

FOR y=0 to myFile2.endOfFile():

scores.append(y)

next y

FOR z=0 to players.length():

scoreList.append(players[z],scores[z])

next z

myFile2.close()

scoreSwap=True

WHILE scoreSwap==True:

scoreSwap=False

FOR n=1 to scoreList.length():

IF scoreList[n-1][1]>scoreList[n][1]:

temp=scoreList[n-1][0]

temp2=scoreList[n-1][1]

scoreList[n-1][0]= scoreList[n][0]

scoreList[n-1][1]= scoreList[n][1]

scoreList[n][0]=temp

scoreList[n][1]=temp2

scoreSwap=True

ENDIF

next n

ENDWHILE

**Trace table for taking the scores and corresponding players from their files and putting them in a 2D array:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **x** | **y** | **z** | **players** | **scores** | **scoreList** |
| 0 | - | - | [FM89] | - | - |
| 1 | - | - | [FM89, RT34] | - | - |
| 2 | - | - | [FM89, RT34, EK57] | - | - |
| 3 | - | - | [FM89, RT34, EK57, CH63] | - | - |
| 3 | 0 | - | [FM89, RT34, EK57, CH63] | [38] | - |
| 3 | 1 | - | [FM89, RT34, EK57, CH63] | [38,45] | - |
| 3 | 2 | - | [FM89, RT34, EK57, CH63] | [38,45,36] | - |
| 3 | 3 | - | [FM89, RT34, EK57, CH63] | [38,45,36,57] | - |
| 3 | 3 | 0 | [FM89, RT34, EK57, CH63] | [38,45,36,57] | [[FM89,38]] |
| 3 | 3 | 1 | [FM89, RT34, EK57, CH63] | [38,45,36,57] | [[FM89,38],[RT34,45]] |
| 3 | 3 | 2 | [FM89, RT34, EK57, CH63] | [38,45,36,57] | [[FM89,38],[RT34,45],[EK57,36]] |
| 3 | 3 | 3 | [FM89, RT34, EK57, CH63] | [38,45,36,57] | [[FM89,38],[RT34,45],[EK57,36],[CH63,57]] |
|  |  |  |  |  |  |

**Trace table for sorting the 2D array in ascending order of score:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **scoreSwap** | **n** | **scoreList[n][0]** | **scoreList[n][1]** | **temp** | **temp2** | **scoreList** |
| True | - | - | - | - | - | [[FM89,38],[RT34,45],[EK57,36],[CH63,57]] |
| False | 1 | RT34 | 45 | - | - | [[FM89,38],[RT34,45],[EK57,36],[CH63,57]] |
| False | 2 | EK57 | 36 | RT34 | 45 | [[FM89,38],[RT34,45],[EK57,36],[CH63,57]] |
| True | 2 | RT34 | 45 | RT34 | 45 | [[FM89,38],[EK57,36],[RT34,45],[CH63,57]] |
| True | 3 | CH63 | 57 | RT34 | 45 | [[FM89,38],[EK57,36],[RT34,45],[CH63,57]] |
| False | 1 | EK57 | 36 | FM89 | 38 | [[FM89,38],[EK57,36],[RT34,45],[CH63,57]] |
| True | 1 | FM89 | 38 | FM89 | 38 | [[EK57,36],[FM89,38],[RT34,45],[CH63,57]] |
| True | 2 | RT34 | 45 | FM89 | 38 | [[EK57,36],[FM89,38],[RT34,45],[CH63,57]] |
| True | 3 | CH63 | 57 | FM89 | 38 | [[EK57,36],[FM89,38],[RT34,45],[CH63,57]] |
| False | 1 | FM89 | 38 | FM89 | 38 | [[EK57,36],[FM89,38],[RT34,45],[CH63,57]] |
| False | 2 | RT34 | 45 | FM89 | 38 | [[EK57,36],[FM89,38],[RT34,45],[CH63,57]] |
| False | 3 | CH63 | 57 | FM89 | 38 | [[EK57,36],[FM89,38],[RT34,45],[CH63,57]] |

This algorithm organises the scores and corresponding users by ascending order which allows these users and scores to be put on the leaderboard in the game. This will let anyone playing the game see who has done best at it and would encourage players to try and get an even higher score so that they can be at the top of the leaderboard. There are separate files used to store the usernames and scores which add a new username and score when someone dies playing the game. The program will also take the scores and usernames in these files and stores them to in two separate 1D arrays before combining these arrays into one 2D array which puts each username with its corresponding score. The text files are the easiest way of permanently storing information on those who play the game and I have also used the easiest way of adding them back into the program using a 2D array. The algorithm then performs a bubble sort on the items in the 2D array to make sure the 2D array stores items in ascending order of score so that they can be shown in the correct order on the leaderboard. A bubble sort is the easiest sort to implement into the program and whilst it is the slowest, the time taken to sort through the whole 2D array should still be too short for the player to notice.

Algorithm 3

**Sub-routine for the player taking damage and dying:**

FUNCTION playerDamage(player.health, username, score, damageFrom, gameOver):

IF damageFrom== “basicWeapon”:

player.health=player.health-10

ELSE IF damageFrom== “harderWeapon”:

player.health=player.health-20

ELSE IF damageFrom== “bigAttack”:

player.health=player.health-40

ELSE IF damageFrom== “constantDamage”:

player.health=player.health-1

ExplosionSound.play()

IF player.health<=0:

FinalExplosionSound.play()

myFile=openWrite(“usernames.txt”)

myFile.writeline(username)

myFile.close()

MyFile2=openWrite(“scoresAchieved.txt”)

MyFile2.writeline(score)

MyFile2.close()

OUTPUT (username+“ got a score of ”+score)

gameOver=True

ENDIF

return gameOver

ENDFUNCTION

**Trace table for the function:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **username** | **score** | **player.health** | **damageFrom** | **gameOver** | **Output** |
| FM89 | 50 | 100 | harderWeapon | False |  |
| FM89 | 50 | 80 | basicWeapon | False |  |
| FM89 | 50 | 70 | constantDamage | False |  |
| FM89 | 50 | 69 | constantDamage | False |  |
| FM89 | 50 | 68 | constantDamage | False |  |
| FM89 | 50 | 67 | constantDamage | False |  |
| FM89 | 50 | 66 | bigAttack | False |  |
| FM89 | 50 | 26 | bigAttack | False |  |
| FM89 | 50 | -14 |  | False | FM89 got a score of 50 |
| FM89 | 50 | -14 |  | True |  |

This algorithm determines how much damage the player should take when they get hit by an enemy weapon or another hazard and it also checks whether the player’s health is below 0. In the game, different enemies will do different amounts of damage and when a player is hit by a certain weapon, the damageFrom variable will can assigned a different value, which determines the amount of health the player loses. For example, if a basic enemy’s laser hit the player damageFrom would equal “basicWeapon” so the player would lose 10 health points but if they were hit by a big explosion, they would lose 40 health points as the value of damageFrom would equal “bigAttack”. Having different enemies deal different amounts of damage means that the player will have to be more wary those stronger enemies and prioritise them in any given wave which increases the amount of skill they need to pass a wave. Moreover, enemies that do more damage could show up more and more in later waves which would help increase the game’s difficulty over time.

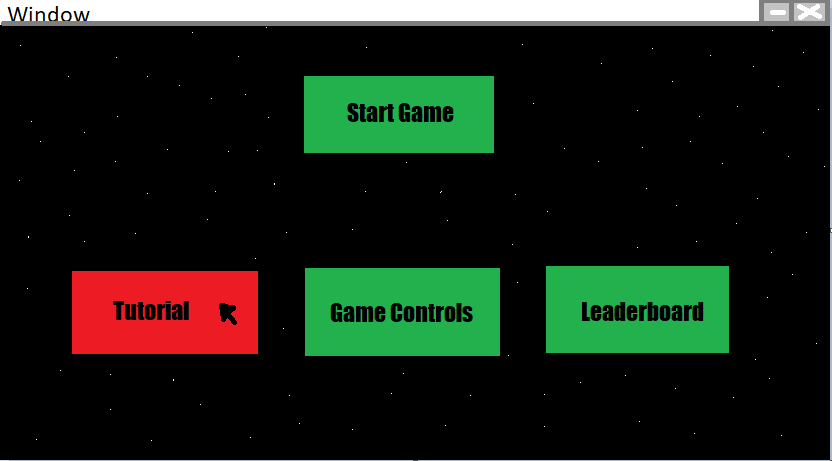
Usability Features

The first usability feature I will use is that when the player is entering the height and width of the screen at the beginning, the program will specify the maximum and minimum heights and widths that can be entered, with the maximum values depending on the resolution of the screen being used. This is so that the heights and widths entered can’t be bigger than the screen itself and can’t be too small which would mean the user wouldn’t be able to dodge enemy attacks. Then, if the user enters an incorrect height or width, the program will tell them they did that and let them reenter the height and width. This is to make sure that when the user enters incorrect heights and/or widths, the program won’t just stop working.

On the main screen, the different icons you can click on will be clearly labelled in a big font about their purpose and when the user hovers over an icon it will change colour so that the user knows they can click on it. It is done this way because it will be easy for the user to click on different sections of game and they won’t be confused as to when they can click on certain icons on the screen. These icons would take the player to different sections which are: a leaderboard, set of controls, a tutorial area and a start to the game depending on the icon clicked.

When a user clicks on a particular section, there will be a back button that lets them return to the main menu in a corner of the screen. This makes navigation between different sections much easier since they can always return to the main screen from any section to go to another one. In addition, the layout and design would remain consistent on each of the sections with the same style boxes being used to represent the icons. These icons will also use the same fonts and will all change to the same colour when the mouse hovers over them. This consistency among the menu’s icons means that the user will always understand what on the screen they can click on to take them to another section of the game and it also means that it will be easy to remember how to access information and perform tasks in the different section as the layout is consistent throughout so even if they are returning to the game after a long period of time, they will understand how to navigate the menus.

Usability will extend to the controls of the game itself with the player moving their spaceship using the up, down, left and right arrow keys. I am choosing these keys because it will be very easy for a user to remember how to move their spaceship up, down, right and left as the movement options match up with the corresponding keys. If the game ends up having a two-player mode and there are two people playing on the same keyboard, then one player will use the up, down, left and right keys to move their spaceship and the other will use the WASD keys to move up, left, down and right respectively. This is because WASD keys are also associated with moving a character so it will be easy to remember how to move the second spaceship in a co-op game. In addition, the WASD and arrow keys are far apart enough on the keyboard that the two players hands shouldn’t collide whilst they are trying to control their own character.



**This is a design for the main menu screen. The icons for the different sections are usually green but the mouse is currently hovering over the tutorial icon so it has turned red to show that it can be clicked.**

Client Feedback

I asked my stakeholders what their initial thoughts of my UML diagrams, algorithms, hierarchy charts and the usability features were, these were their responses:

Client Response:

“Clear and practical features such as having buttons lighting up when hovered over. The algorithms are efficient and concise, and the program is shown to be well structured in the use case diagram and the hierarchy charts show that the problem has been decomposed well”

Stakeholder 1:

“I think the hierarchy charts are very clear, and the UML diagrams give a good overview of the project as a whole. The algorithms are really detailed and show how the program will be structured. The usability features are carefully considered and will make it easy for all users to access the game.”

Stakeholder 2:

“The hierarchy charts are easily understandable. Algorithms are detailed and do a good job at showing how the game will be coded. The UML diagrams are logical and clear and the usability features are simple and make the games features easily accessible.”

Key Variables and Classes

**Variables:**

|  |  |  |
| --- | --- | --- |
| **Variable** | **Data Type** | **Use of variable** |
| Fps | Integer | It will store the number of frames the game will run at per second. The larger the value stored by the Fps variable, the faster the game will run so if the game has multiple difficulties then higher difficulties could use a larger Fps variable so that everything in the game happens faster which will make enemies and hazards harder to avoid. |
| gameOver | Boolean | When the player begins the game from the main menu, the gameOver variable is set to True and when the player dies it is set to False which causes the game to go back to the start menu. Each time the player restarts the game the gameOver variable is set to True again and each time the gameOver=True, the program will reset several other variables and sprite groups so that the main structure of the game is retained whenever someone restarts the game. |
| noEnemies | Integer | It stores the number of enemies that will spawn in each wave. This will increase by a given number each wave to make the game harder over time. There will also be different versions of this variable for different enemy types as later waves will have different numbers of different enemy types spawning onto the game. |
| Score | Integer | The score of the player playing increases whenever they destroy an enemy spaceship, with the enemies awarding more points if they are harder. Defeating bosses will also give the player the more points than killing enemies. The final score of a player when they die will be stored by the game in a text file and the top five scores will be stored on the main menu of the game. |
| gameRunning | Boolean | The main piece of code for the program will be inside a while loop. So, the whole program will run while gameRunning is True. This variable will never be set to False because that would break the game itself. |
| Username | String | The player enters a username before they begin the game. This username is then stored alongside their number of points in the text file when they die so that players will know how well other people did at the game. It will also allow those players with the top five scores to be stored in game. |
| Phase1/2/3/... | Boolean | There will be different variables representing Phase1, Phase2, Phase3 etc. When Phase1 is True, the game is in its first phase with only certain enemies and only certain numbers of enemies. After the first boss is beat, the game makes Phase1 False and Phase2 True which means that the game is now in its second phase which has new enemy types and more of them. This then repeats with Phase 3, Phase 4 and so on. |
| Height/Width | Integer | When the user starts the game, before they reach the main menu, they enter the height and width of the screen. The user enters these details themself because otherwise the game screen might not fit onto certain computer screens like laptop screens. |
| bossTime | Boolean | When a boss enters onto the screen this variable becomes True and whilst it is True other enemies, and some hazards cannot spawn in the game. This is because it allows the player to directly fight the boss which is more enjoyable than fighting the boss and several other enemies at once. |
| ScoreArray | 2D Array | This will hold all the scores stored in the text file as well as the username of the player with those scores. They will be ordered in ascending order and this list will be presented when the current player wants to look at the leaderboard. |

**Classes:**

|  |  |
| --- | --- |
| **Class** | **Use of Class** |
| Player Class | This class stores the attributes and methods that the player has and uses. There are attributes for the player’s speed, ammo, image and amount of health amongst other things and many of these attributes will vary depending on the interactions with enemies and hazards. The player class includes methods that allow the player to move up, down, left and right as well as being able to call the classes of the player weapons so that the player can shoot lasers or bombs at the enemies. |
| Enemy Class | This class gives attributes for the enemy’s starting co-ordinates, speed and image used. The attributes for the coordinates will be randomised at some point on the right-hand side of the screen. Its method makes it moves left constantly until it is either destroyed or goes offscreen and it also gives the ship a small chance of calling its own Laser class to shoot at the player ship. There will be different classes for different enemies but since they share similar attributes and methods inheritance could be used here to make cut down on the code that needs to be written. |
| Player Weapons Classes | There will be separate classes for player lasers and player bombs and these classes are called into the player class when the player presses certain keys on the keyboard. Both classes will have attributes for the starting coordinates, the image used and their speeds with the starting coordinates either being at the bottom or the right of the player ships depending on whether it’s a bomb or laser. The bomb’s method will make it move down and the laser’s method will make it move right. |
| Meteor Class | This class has attributes for its y and x velocity since its velocity along the x axis is randomised to allow meteors to either move right, left or straight down when moving downwards. This will make meteors more difficult for players to dodge so they need to be more skilled in order to do that. Its methods make it move downwards from the top of the screen and make the meteor image rotate after a given number of frames which will make the meteor look more believable. |
| Laser beam Class | This class randomises its starting y coordinate so that it either moves across the top of the screen or the bottom of it. It also includes attributes for its size, image used and speed. This speed could be increased over time so that the laser beam becomes harder to avoid the further you go into the game. Its method just makes the laser beam move right across the screen in accordance with its preset speed. |
| Boss Class | Each boss will have its own class but since they will share many attributes and methods, inheritance could be used here in a similar way to how it would be used with the enemy class. Each boss will have methods to perform their different attacks as a boss needs multiple attacks in order to stay interesting for the player. Each method will have a random chance of being triggered every frame so that attacks can stay varied as well. |

Validation

There are some user inputs that need to be validated otherwise the game won’t function correctly. The inputs that need to be validated are:

* Username- Each person playing the game will need to enter in a username, so they are identifiable on a leaderboard that another player looks at. Therefore, a presence check will be used to make sure that the something is entered as a username and there will also be a length check to make sure the username entered isn’t too long, which would have made it hard to fit on a leaderboard.
* Height and Width- The height and width of the screen entered can’t be smaller than a certain amount otherwise it would become impossible for the user to avoid the enemy lasers and other hazards. This means the game will check that the height and width are big enough and, if I can, the game will also check that the height and width aren’t bigger than the computer screen itself because otherwise the game won’t fit on the computer screen.
* Player Movement- The game needs to make sure the player can’t move off the screen, like going above or below it because otherwise they can easily avoid enemy attacks and they also won’t be able to see their character. The program will constantly check that the user’s x and y coordinates and if they are trying to move in each direction but one of the relevant coordinates is too big or too small then they won’t be able to move in that direction. For example, if they are trying to move left but their x-coordinate is already 0 then they can’t move any further left.
* Shooting Lasers- The lasers start with a certain amount of ammo and when the player shoots lasers they use some of that ammo from a player attribute storing their amount of ammo. Therefore, the game needs to check if the player has any ammo left when they press the key to shoot or else, they won’t be able to shoot any lasers. There is a limited amount of ammo to make sure players aren’t just always firing lasers all the time as that would make beating enemies too easy and it also encourages the use of bombs as well. In addition, it encourages players to shoot at meteors which can drop extra ammo as a powerup.

Testing

Iterative Testing

During the program’s development, I will be repeatedly testing the code and its variables when I finish a section of the program. I will continue testing that section until I have removed all errors and until that section achieves the part of the primary success criteria it is needed. This process will then be repeated for every other section of the program until I have a finished game that achieves all its primary success criteria which are its functional requirements. After this game is complete, I will start making and testing new sections of code to achieve my non-functional requirement and the same iterative testing will be used for each section of code for non-functional requirements. The iterative testing will also use whitebox testing to make sure that that section of the program is functional as well as making sure the code is written as efficiently as I am able to write it.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Sub-routine tested in** | **Variable being tested** | **Test Data** | **Type of Data** | **Expected result** | **Actual result** |
| moveCharacter() | keyPressed | “up” | Valid | Moves player character up by 10 pixels. |  |
| moveCharacter() | keyPressed | “down” | Valid | Moves player character down by 10 pixels. |  |
| moveCharacter() | keyPressed | “left” | Valid | Moves player character left by 10 pixels. |  |
| moveCharacter() | keyPressed | “right” | Valid | Moves player character right by 10 pixels. |  |
| moveCharacter() | keyPressed | “f” | Invalid | Player won’t move |  |
| shootLaser() | keyPressed | “z” | Valid | Laser is shot from the front of the spaceship |  |
| shootLaser() | keyPressed | “n” | Invalid | A laser isn’t shot. |  |
| dropBomb() | keyPressed | “x” | Valid | A bomb is dropped underneath the spaceship. |  |
| dropBomb() | keyPressed | “m” | Invalid | A bomb isn’t dropped. |  |
| - | Height | 1600 | Valid | Sets the screen’s height to 1600 pixels |  |
| - | Height | 1920 | Borderline | Sets the screen’s height to 1920 pixels |  |
| - | Height | 200 | Invalid | Should ask the player to reenter the height. |  |
| - | Width | 800 | Valid | Sets the screen’s width to 800 pixels |  |
| - | Width | 1080 | Borderline | Sets the screen’s width to 1080 pixels |  |
| - | Width | 100 | Invalid | Asks the player to reenter the width |  |
| mainMenu() | username | “FM89” | Valid | Accepts FM89 as the username and starts the game. |  |
| mainMenu() | username |  | Invalid | Asks the user to enter a username again |  |
| mainMenu() | username | “oasfpjafisadffadi” | Invalid | Asks the user to enter a shorter username |  |
|  | bossTime | True | Valid | One boss should spawn into the game |  |
|  | bossTime | False | Valid | A boss won’t spawn into the game. |  |
|  | gameOver | True | Valid | The game returns to the main menu |  |
|  | gameOver | False | Valid | The game will continue. |  |
|  | PlayerScores 2D array | The players and their scores. | Valid | The players and scores should be arranged in the 2D array in ascending order of scores. |  |

All the keyPressed variable inputs tested within the moveCharacter() method are used to test whether only the up, down, left and right key can be used to move the player in those directions. If the player doesn’t press one of those keys, then the spaceship shouldn’t move. Similarly, the keyPressed variables for the player’s attacks are tested to make sure they only work with the keyboard keys “z” and “x” and not with any other keys. The values entered for the height and width of the screen are used to make sure that the height or width entered can’t be bigger than the height or width of the monitor itself and can’t be too small which would make the game unplayable as the player wouldn’t be able to dodge enemy attacks. The username variable is tested to make sure that it isn’t accepted if it is too long or if nothing is entered for the username because otherwise it either wouldn’t fit on the scoreboard, or you wouldn’t be able to tell who got a particular score. Testing the bossTime variable is done to make sure that the boss only spawns in when this variable is true because otherwise there would be bosses spawning in regular waves which would make those waves too harder. Similarly, the gameOver variable is tested to make sure that it only becomes True when the player dies and that when it is True the game will return to the main menu. This is because if this wasn’t checked and gameOver became True for another reason, then the game might just suddenly end for a user and if gameOver didn’t take the player back to the main menu then they wouldn’t be able to restart the game when they died and would have to exit the game instead. Finally, the order of the playerScores array is checked to make sure it is in ascending order because otherwise the leaderboard wouldn’t be able to present the scores in any order for the player so it would be harder for the player to find the person with the highest score on the leaderboard.

Post-Development Testing

|  |  |
| --- | --- |
| Other actions to test | Does it work? |
| Moving mouse over an icon in the main menu should change the icon’s colour. |  |
| Clicking on an icon should take you to the section of the menu the icon points to |  |
| When a player attack collides with an enemy, the enemy should lose a fixed amount of health. |  |
| When an enemy loses all its health it should be destroyed. |  |
| When a player is hit by an enemy attack or other hazard, they should lose health |  |
| Different enemy attacks will do different amounts of damage |  |
| When the player loses all their health, gameOver is set to True |  |
| The second wave will come in at a fixed time after the first one, the same goes for the third wave, fourth wave etc. |  |
| Each wave increases the number of enemies or introduces new enemies. |  |
| The meteors have a small chance of dropping a powerup when destroyed. |  |
| When a boss enters the screen, all other enemies stop spawning. |  |
| When a boss is destroyed, the game should move onto the next phase with new enemies and hazards. |  |
| The leaderboard is updated when the game ends. |  |
| When the player dies, their username and score are stored in their own files. |  |
| Increase the player score by an amount in line with the difficulty of a given enemy when it is destroyed. |  |
| When the player gets a powerup it gives them a certain ability like extra ammo or health. |  |
| The controls work for two players. |  |

These are the additional tasks that don’t necessarily have an input tied to them that must be completed for the game to be considered finished. These are needed so that all the functional requirements of the success are completed for the game. The first two actions that are tested test how easily navigable the main menu of the game is which lets the user know how interactive the UI is. The next two control how player weapons damage enemies and when enemies are destroyed which is important in terms of knowing when players get points. Then the following three actions test how the player is damaged differently depending on the attack and tests that the game ends when the player loses all their health. The actions testing the time between waves and the enemies in waves are used to make sure that the waves become more difficult over time. The actions testing meteors and powerups test that the powerups will always boost the player in some way and that they will spawn from destroyed meteors so that they are accessible in game. The actions to test concerning player scores check whether the player scores are stored permanently after the game finished and test if the leaderboard is updated so that the user can be how well they did after a given game. It also tests to see if harder enemies give the player more points so that when the player lasts a long time in game, they will be able to show that through the number of points they get. Finally, making the controls work for two players means that the game will able playable in co-op.