

# Faculty of Science and Technology

**2018/2019**

**Level 4**

**Object Oriented Games Programming**

# 2D Video Game Program

**Analysis, Design, and Implementation**

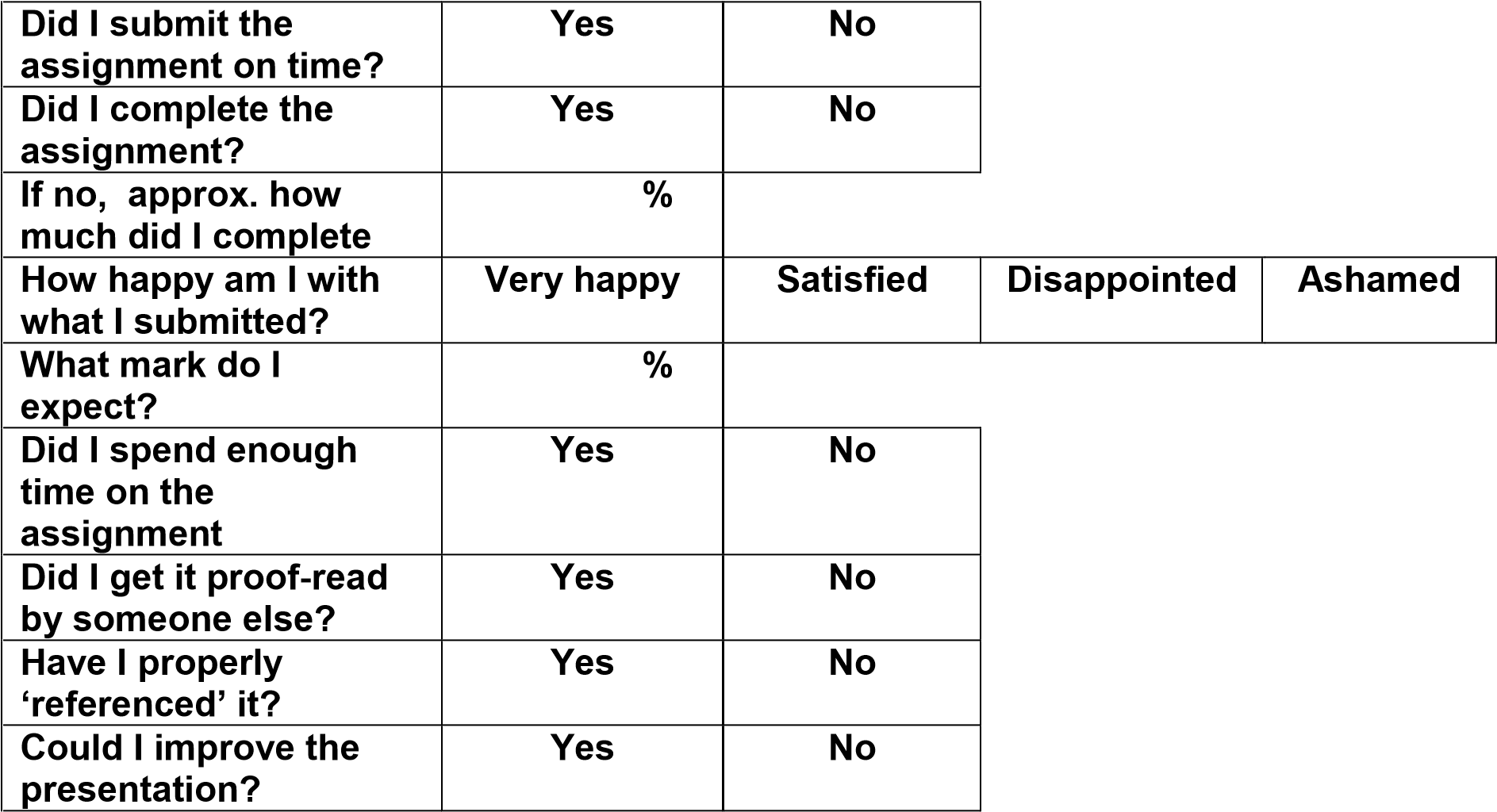
**Report Template**

# 1. Self-Assessment of Performance (Marks: 5%)

**Tutor :** Andrew Watson

|  |  |
| --- | --- |
| **Student’s Id** | **S5111564** |

**Indicate the appropriate response:**



65

**Answer the following questions:**

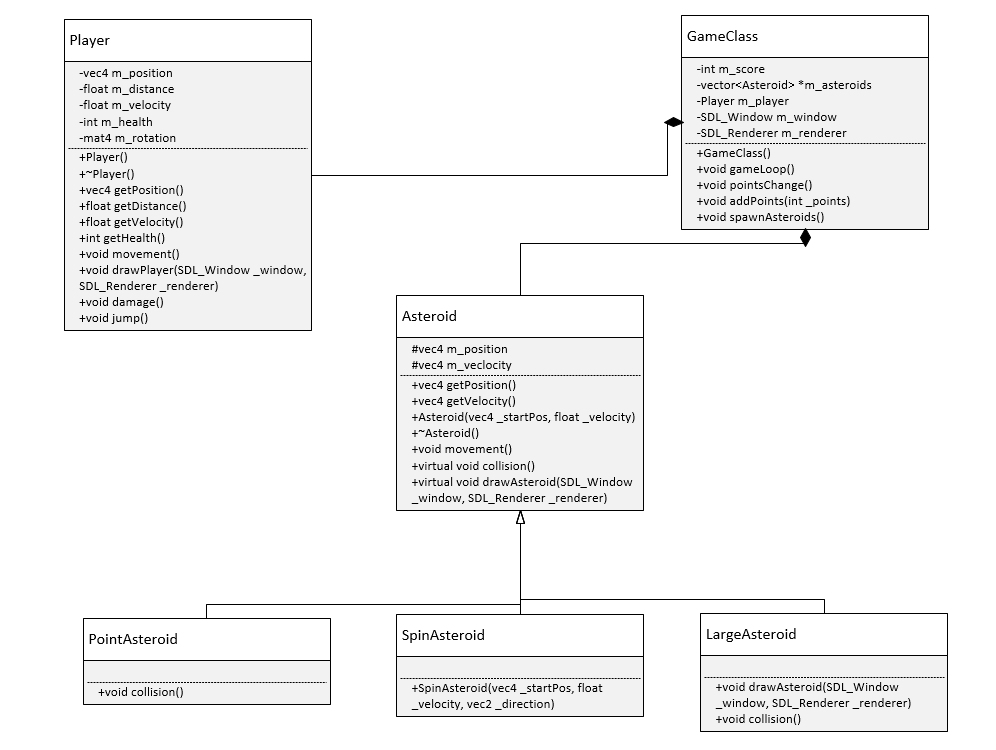
|  |  |
| --- | --- |
| **The best part of my performance was:** | **i think the mechanics of my game are well polished and the game itself is relatively fun** |
| **The worst part of my performance was:** | **The sprites and visuals of my game leave a lot to be desired as I’m not an artist myself. I created simple sprites with pixel art in Photoshop which is the limit of my ability** |
| **One way in which I could improve the content of my assignment is:** | **I could work on adding more depth to the game by creating different asteroid types and ways to control your planet. I could also add some more menu options and a death screen** |
| **One way in which I could improve the presentation of my assignment is:** | **I could have made use of more diagrams in my report to help explain the logic of my program** |
| **One thing I will do to improve my performance in my next assignment is:** | **Now my abilities have improved I could attempt a more complex game than I did in this project using some of the techniques I have learned** |
| **Another thing I will do to improve my performance in my next assignment is:** | **I should make more use of diagrams and flow charts to explain my program’s logic and go into more depth about each module** |

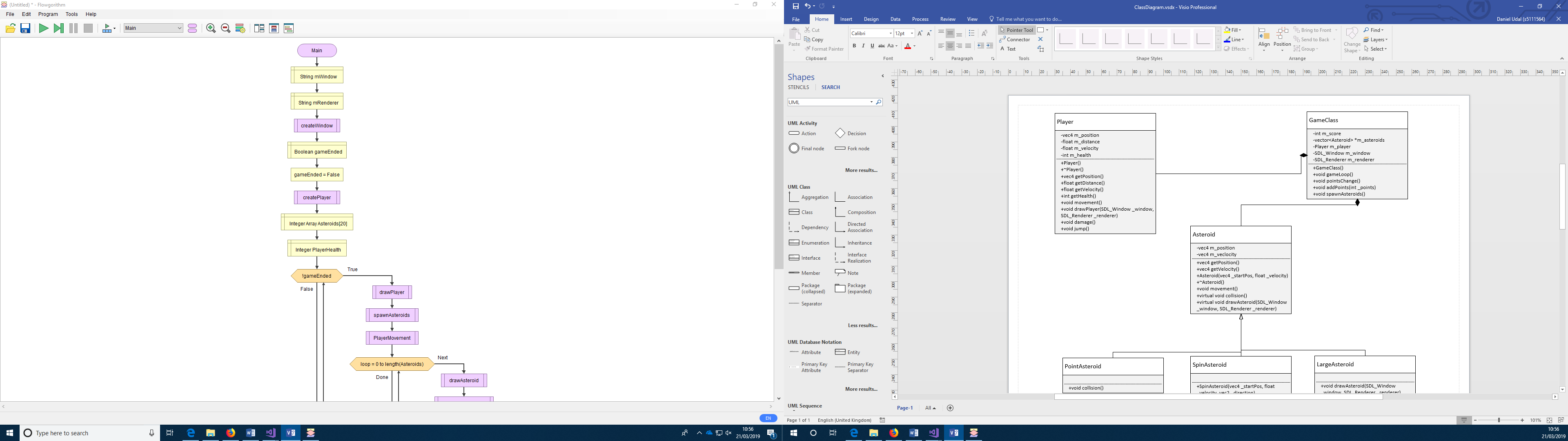
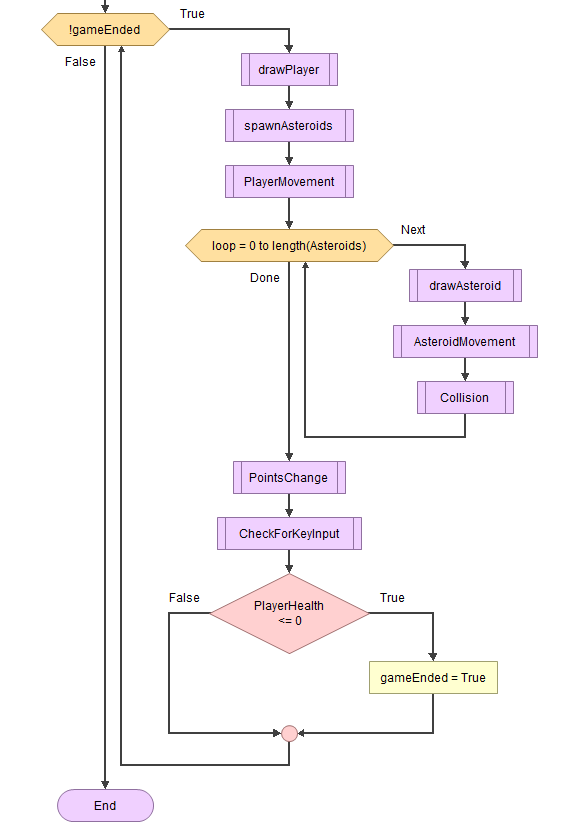
# . Game Concept (Marks: 5%)

I’m making a space survival arcade style game where the player plays as a planet orbiting a star while asteroids come from random directions and fall towards the star. The player’s goal is to survive as long as they can to achieve a high score. The player’s planet can be made to move away from the star by pressing a button in order to avoid asteroids. If the player is hit by an asteroid their planet will slow down and if they slow down too much they’ll fall into the star and lose. There will be 3 kinds of asteroids that can act in different ways for the player to avoid, a basic asteroid that simply flies into the sun, an asteroid that starts with some velocity perpendicular to other asteroids and a larger asteroid that is more difficult to avoid. There will also be an asteroid that gives the player bonus points if they hit it. The player will gain more points for surviving with a lower speed than a higher one and for being closer to the star. As the game goes on the star will take in asteroids to gain mass, increasing its force on the player’s planet which increases the minimum speed the player needs to survive. The game will also begin to spawn asteroids with gradually increasing frequency so the game becomes more challenging over time. The player’s current score will be displayed inside the star.

# . Analysis (Marks: 40%)

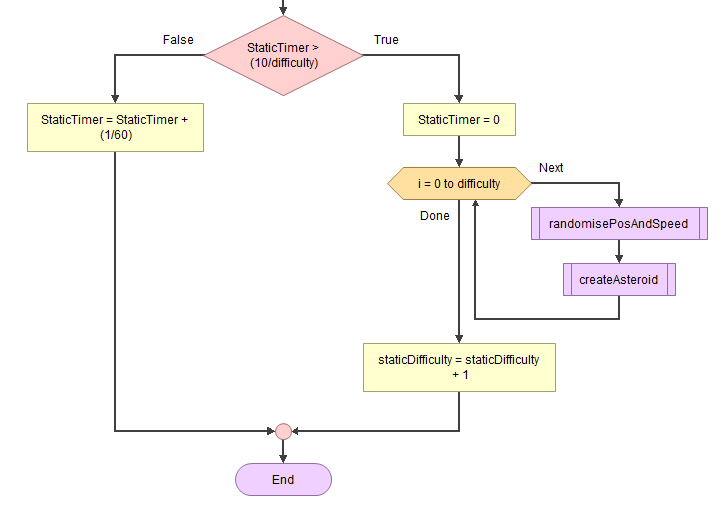
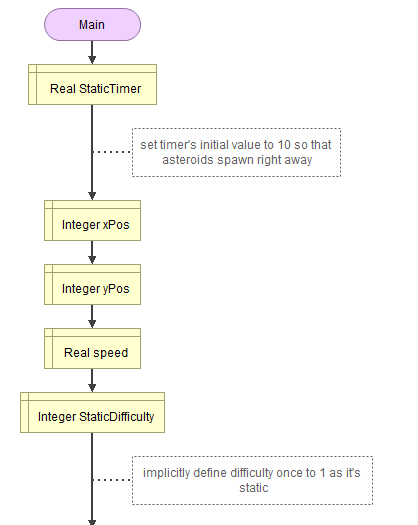
The game breaks down into 3 major classes for its function, asteroids, the player and the game class. There are 4 types of asteroids with the base class serving as one and the other 3 inheriting from it as shown in the UML class diagram below. The game class will encapsulate everything that happens in the game and will handle all game processing. This class has a player contained within it and some number of asteroids. This is shown by the composition line connecting it to both. It contains a vector of pointers to the asteroid class which will allow it to point at that class and any derived from it. This vector will contain every asteroid in the game and I will use its length to iterate through each asteroid in processing. The game class will also contain the SDL window and renderer and use them to create the window during initialisation. These will then be passed into the draw functions in player and asteroid. In asteroid the draw and collision functions are virtual because they need to be overridden in derived classes. In spin asteroid the constructor is rewritten as it needs to be initialised differently to other asteroids. When it’s created the base constructor will be called and then the new constructor in the derived class.

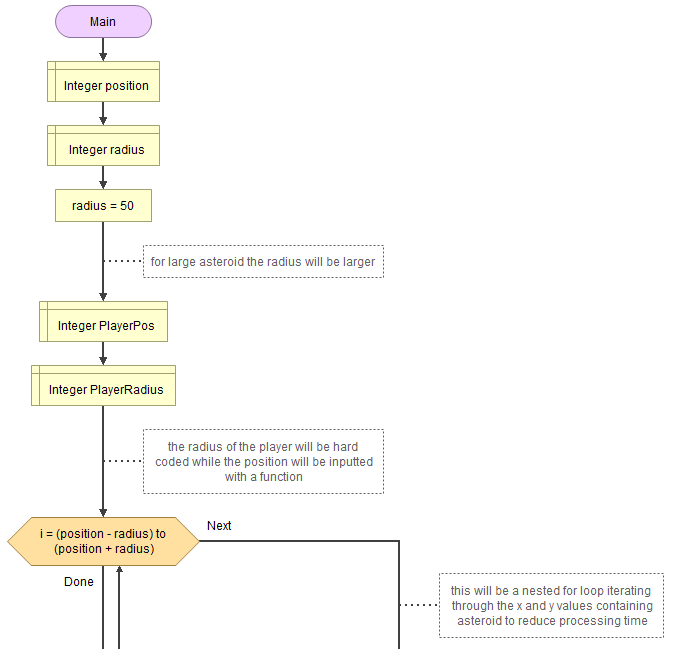
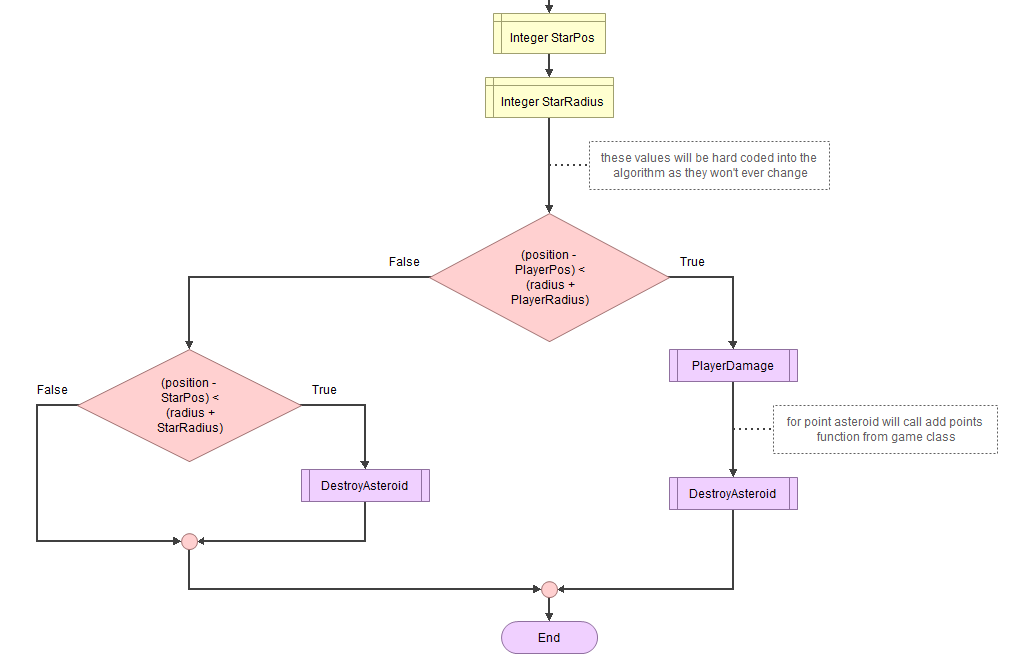




This flow chart demonstrates the primary game loop contained in the game class that will handle all the processing of the game each frame. The start of the flow chart before the loop shows the initialisation that will happen in the game’s constructor before the loop begins. Due to flowgorithm m\_window and m\_renderer were renamed to mWindow and mRenderer and were declared as strings. Asteroids were declared as an array of integers. Every game loop the player is drawn and its movement is processed while the new asteroids are spawned. Then for each asteroid they are drawn, their movement processed and then they check if there was a collision with the player. If there is, the collision function will handle what happens then returns to the game loop. After that the game object processes the change to the points then handles any input from the player. Finally it checks if the player’s health hit 0 and if it did it changes the gameEnded variable to true thus ending the game loop and ending the game.

*In order to fit both parts in the bottom part is before the higher part in logic*

The flow charts displayed here demonstrate the logic of the spawn asteroids function in the game class. The variables timer and difficulty will be declared as static so that they retain their value through each call of the spawn asteroids function. These variables will determine if asteroids should be spawned and how many need to be. This will prevent asteroids being spawned every frame and make the difficulty scale up as the game goes on by increasing the number of asteroids and reducing the intervals between spawns. Specific values will need to be tested and fine-tuned during implementation in order to balance the game.



Above is a flow chart demonstrating the logic of the collision algorithm in asteroid. For the asteroid in question the algorithm will test to see if the distance from centre to centre is less than the sum of their radii. If this condition is true then the asteroid and player must be intersecting. I’ve chosen to allow the collision system to be slightly lenient by using less than rather than less than or equal although this will rarely be noticed by the player. If there is a collision the function to damage the player will be called and the asteroid will be destroyed. If there isn’t a collision with the player then the algorithm will test for a collision with the star in the centre and if there is the asteroid will also be destroyed. If the asteroid in question is a points asteroid a function to gain points will be called instead.

The other functions in the game without logic analysis are the getters, the drawing functions, the movement functions, the simple constructors that simply set all the values for the instance, the destructors, and the player’s damage and jump functions. The getters don’t warrant any analysis as their logic is just to return a single value. The movement functions for the asteroids and player will just be a multiplication of the position by a translation and rotation matrix respectively and as such their logic will be only 1 or 2 lines of code. The constructors for the classes don’t make use of any logic and just set instance variables for the objects and the destructors perform any necessary actions to delete an object, which I expect will be very few. The player’s damage function will only subtract 1 from the player’s health and the jump function will just multiply the player’s position by a translation matrix and add a scalar value to their distance variable. The drawing functions will simply use bitmap images in the files to create their images so will only contain a few lines of code.

# . Design (Marks: 40%)

## Complete this section using Arial font 12pts

Void gameLoop()

{

SDL\_Event input

While(!gameEnded)

{

If (pollEvent = 1)

{

Switch(input)

{

Case spacebar : m\_player.jump()

Case esc : pause()

Case SDL\_Quit : endgame()

}

}

M\_player.movement()

For (int I = 0; I < size(m\_asteroids); i++)

{

M\_asteroids.at(i)->movement()

M\_asteroids.at(i)->collision()

} *multithreading optimization could be used if needed and there is time*

This->spawnAsteroids()

This->drawWindow()

WaitForFrame()

}

}

This is the primary loop for my game which handles the events, spawns any new asteroids needed, calls the movement functions for everything in the game and calls the draw function each frame.

void pause()

{

SDL\_Event input

//display pause screen

while (!unpause)

{

while (SDL\_PollEvent(&input))

{

switch (input)

{

case escape: unpause = true

case resumeClicked: unpause = true

//additional cases for options/quit

}

}

}

}

This is the pause function for my game which will be called when the player presses escape. It stops the processing of the game until complete and finishes when the player triggers an event to end it. This will allow for a pause screen in my game

Int collision(player)

{

If (distance between centres < sum of radii)

{

Return 0 //for player collision

}

If (distance between centre and start centre < sum of their radii)

{

Return 1 //for star collision

}

Return -1 //for no collision

}

This is the collision function for the asteroids. It first tests the distance between the asteroid centre and the player’s then returns 0 if the sum of their radii is less than the distance as a collision has happened. It then does the same with the star so that asteroids are destroyed when they collide with the centre star. Finally it returns -1 if no collision is detected.

My original plan was to perform the processing inside the collision function however I found it made more sense to return the type of collision and handle it in the game loop.

At this point in development making a spin asteroid work seems too complex to implement until after primary functionality is made

void GameClass::spawnAsteroids()

{

if timer > 5.0f

{

for (0 to score mod 500)

{

angle = random float from 0 to 360

spawnPoint x value = 300 \* cos(angle)

spawnPoint y value = 300 \* sin(angle)

speed = random float from 10 to 30

type = random integer from 0 to 9

if type < 4

{

Create base asteroid

}

else if type < 8

{

Create large asteroid

}

else

{

Create point asteroid

}

}

Timer = 0

}

}

This is the function in the game class to spawn asteroids at random positions and with random velocities. A random angle relative to the centre is generated and the polar coordinates are converted to Cartesian to be used in vectors. The magnitude of their position is hard coded to be 300 so the asteroids spawn on an invisible circle around the centre. Then a random number from 0 to 9 is generated to create a percentage chance for each of the asteroid types. Base and large asteroids have a 40% chance while point asteroids have a 20% chance. Finally the timer is reset to 0 in order to wait for the next spawning time.

During development I decided that the game needed more control so I added a dive function triggered by left shift. It does the same as the jump function except the operators are reverse, i.e. distance from star is reduced, and velocity is increased. This gives the game a lot more depth to its gameplay.

Its logic is shown here

void jump()

{

m\_distance + 20 / 60

m\_position + unit vector from centre to star \* 20 / 60

m\_velocity - 15 / 60

}

And dive has the reverse operators

# 5. Testing and Conclusions (Marks: 10%)

I tested my game many times through development as I made it in order to make sure each section was working properly as I went. When testing the game I just ran the game in debug to either check it didn’t crash or to test the gameplay by trying the game. As I developed my game I had to remove or change a few of my initial ideas such as removing the asteroid with a spin or performing the collision processing outside the collision testing function. Overall the game is successful at delivering on my initial ideas for the most part as the core gameplay is how I wanted it to be. i did have to remove a major mechanic and adjust my initial ideas for how the gameplay will act as I developed the game, specifically the velocity mechanics where I had to make it less dynamic for ease of coding. During development some issues I encountered were the images being contained in a white box. I made several unsuccessful attempts to fix this by making the background transparent and eventually I had to compromise and just make the background black in the bitmap. I also had an issue where the input was always staying the same at 768 so the test for which key wasn’t working as intended. After some research I found that the type function I was using was incorrect and once I changed it to the correct one my input worked properly. I also encountered a problem with the increasing velocity where the player’s velocity increased to a ridiculous speed around the star. This was caused by the rotation function adding extra rotation to the matrix instead of replacing it. I solved this problem by using an identity matrix as an input instead of the rotation matrix.

# 6. References

## Complete this section using Arial font 12pts

***This section should be used for all referenced materials: Web sites, tutorials, Books, Articles, Source code, Acknowledgements.***