Introduction to C++ Task 4: Design Document

Game: Asteroids
Version: 1.0
Developer: Itai Doron
Version Control: https://github.com/Spoegur/Asteroids.git

Game Overview:

This game is a take on the retro game “Asteroids”. The players pilot a spaceship through a 2D top down space filled with differently sized asteroids. The aim of the game is to survive for as long as possible whilst avoiding asteroids and gaining points, the player gains points by destroying differently sized asteroids and once the arena is free of asteroids the player advances to the next round.

Gameplay Mechanics:

Player Controls:

The player navigates the arena using the arrow keys or WASD keys with left and right keys controlling clockwise/anticlockwise rotation, up keys controlling forward movement and down keys slowing the player down. The player can also use the spacebar key to shoot lasers from the front of the ship, said lasers are what the player must use to destroy asteroids.

Movement and Player Lives:

Other gameplay mechanics also include a momentum based movement system which uses drag and acceleration to insure more interactive and strategic movement gameplay. As well as a basic health system in which the player starts the game with 3 lives, and each collision with an asteroid subtracts a life until game over.

Scoring/Points System:

Another mechanic is the scoring system. The system consists of a score counter in the top right of the screen which steadily increases whenever an asteroid collides with a laser. Different asteroids also give different rewards e.g. the largest asteroids are worth 20 points, medium asteroids are worth 50 and the smallest asteroids 100

Asteroid Splitting:

The final game mechanic worth mentioning is the Asteroid splitting function. In which the large and medium sized asteroids will split into 2 smaller asteroids when they collide with a laser or the player. Also worth noting the smallest asteroids do not split instead they are just destroyed.

Game Progression:

Game progression works by using rounds to increase difficulty; whenever the player clears a stage of all asteroids, they progress to the next round in which a greater amount of asteroids than the round before are spawned at the beginning of the stage. The amount of asteroids spawned are determined by [$x + 4 = y$] where x is the current round and y is the number of asteroids. One exception to this is once the player surpasses round 6 the number of asteroids spawned doesn't increase.

Graphics:

The graphics library I used for this project was Raylib C++, and the textures I used for entities were all resized and recolored assets from Kenney Simple Space and Kenney Space Shooter redux. Assets used include:

- Ship texture (Simple Space): ship_J
- Laser texture (Simple Space): effect_purple
- Destroyed Ship texture 1 (Space Shooter Redux): playerShip1_damage1
- Destroyed Ship texture 2 (Space Shooter Redux): playerShip1_damage2
- Destroyed Ship texture 3 (Space Shooter Redux): playerShip1_damage3
- Large Asteroid Texture (Space Shooter Redux): meteorGrey_big1
- Medium Asteroid Texture (Space Shooter Redux): meteorGrey_med1
- Small Asteroid Texture (Space Shooter Redux): meteorGrey_small1

Algorithms and Data Structures:

Data Structures:

- **Classes:** User defined data structure used to differentiate and store specific data members, member functions and other data structures. Classes are by default private.
- **Struct:** Same as a class except its default state is public.
- **Vector:** A dynamic array (vector) used for storing class member functions e.g. asteroids vector used for storing existing instances of the Asteroid class.
- **Vector2:** A Raylib exclusive 2D vector struct for storing x,y values e.g. position, size, speed.
- **Rectangle:** A Raylib exclusive rectangle struct used for storing position coordinates (x,y) and size variables (Width, Height).
- **Texture2D:** A Raylib exclusive struct used for storing textures, it also contains the height and width values of the stored texture.

Algorithms:

- **Tick():** Gameplay loop which uses the while loop statement to cycle through the necessary functions to run the game e.g. Update() and Draw().

- **Collision Detection:** A Basic bounding circle collision detection algorithm that checks if the distance between the centre of two objects is less than the sum of their combined radius's and return a boolean based on the result
- **Update():** A function that handles the movement of objects using position, speed and acceleration values. As well as any other statement or function that changes the state of an object e.g. collision detection and screen wrapping.
- **Screen Wrap:** A basic screen wrap algorithm that checks whether an object's position is greater than the screen border plus the object's texture height. The object's position is then inverted to the corresponding opposite side of the screen.
- **Ship Rotation:** An algorithm which rotates the ship left or right by multiplying the ships rotation speed by the current delta time between the last drawn frame, and the most recent frame drawn and then adding that sum to the ships rotation.
- **Ship Acceleration:** An algorithm responsible for increasing the player's acceleration. It does this by multiplying the ship's drag speed by 4 and adding the sum to the player's acceleration until they reach the max speed.
- **Ship Drag:** Ship drag works by subtracting the player's acceleration by how long the ship has been moving for. It does this by starting a stopwatch whenever the player's acceleration is greater than 0 and restarting the stopwatch if the player's acceleration reaches 0.
- **Asteroid Splitting:** An algorithm that handles the splitting of asteroids when they collide with a bullet or the player. Large asteroids split into 2 medium asteroids, medium asteroids split into 2 small asteroids and small asteroids are just destroyed. The algorithm works by changing the hit asteroid's size and rotation, then creating a copy of said asteroid with all the same values but a different rotation and speed again.