[Asteroids On A Budget]

Use Case Description

The **Asteroids on a Budget** game is an arcade-style space shooter where the player controls a spaceship model and navigates through a space field of asteroids and aliens. The player's aim is to survive, and shoot and destroy asteroids. As the asteroids are shot, they break apart into smaller fragments which creates an ongoing challenge. The game keeps track of the player's health, score, and survival time, and provides a game-over screen, with the ability to replay, when the player's health reaches zero. The user interacts with the game through keyboard and mouse controls, and individual graphics, e.g., player models, asteroids, etc., are rendered through each respective class, and the UI is rendered through the main file.

Class List

1. Player: Representing the player itself as a player-controlled spaceship with attributes such as position, health, speed, shooting mechanics, and rendering. The movement and rotation of the spaceship will be based on user input, along with shooting the projectiles. Player will also manage the collisions between the player model and asteroids or enemy projectiles.

2. Projectiles: An abstract base class for all projectiles in the game, e.g., player projectiles and enemy projectiles. It will be responsible for defining common attributes like position, speed, and damage, while also declaring virtual methods for movement, rendering, and collision detection for each instance of a projectile.

3. PlayerProj: Derived from the projectiles class and represents the projectiles shot by the player and checks for collisions to destroy the enemy. PlayerProj manages the speed of the projectile, the direction, and rendering, along with the lifetime of the projectile, i.e., disappearing when off-screen or when hitting another object.

4. Enemy: An abstract base class for all enemy objects (e.g., asteroids, aliens, blaster aliens, etc.,). It will define common attributed such as position, speed, and health, along with virtual methods for movement, rendering.

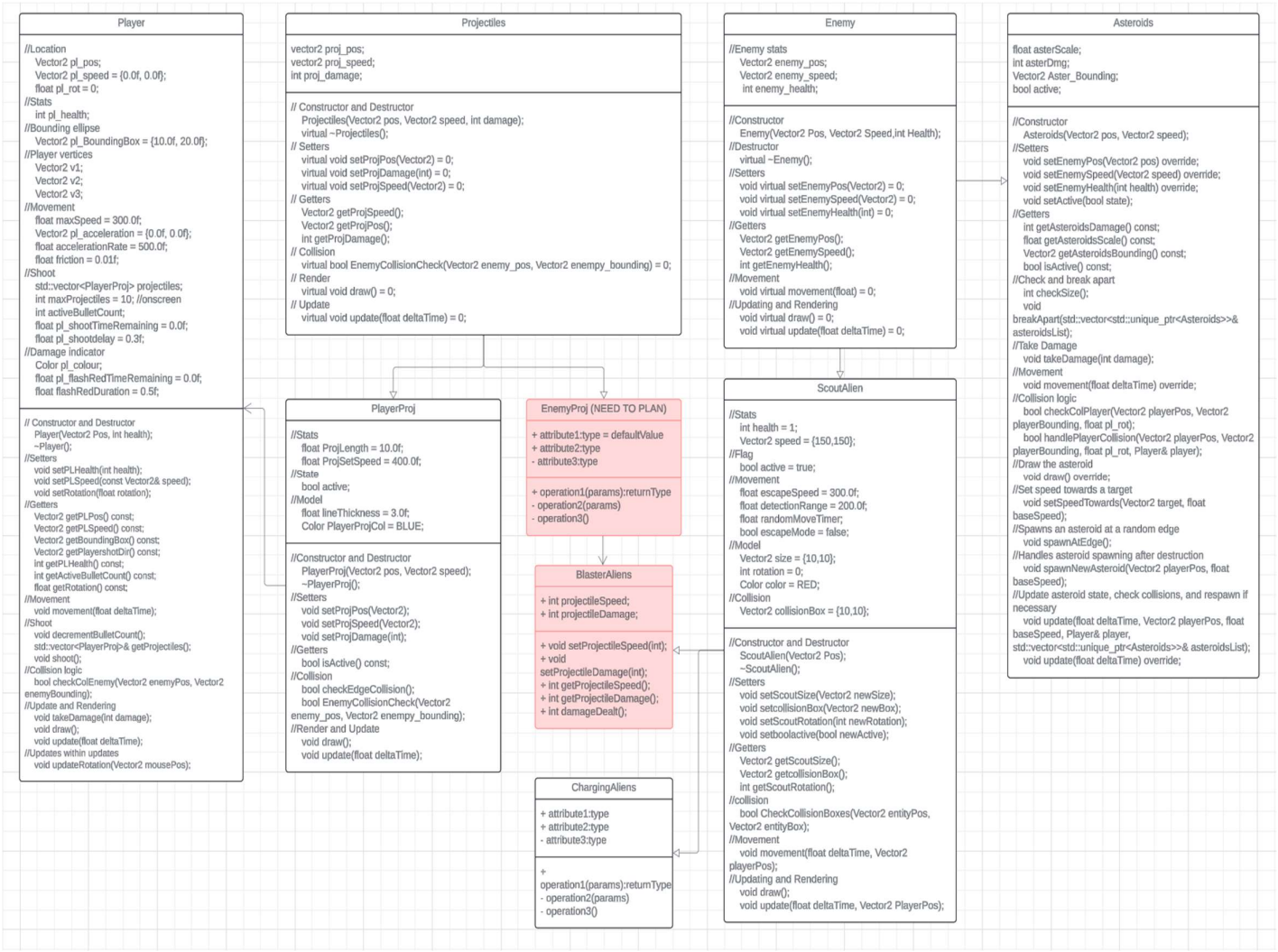
5. Asteroids: Derived from the enemy class and represents an asteroid in the game, differing in sizes and behaviours depending on bounds and scales. Asteroids will be responsible for movement of an asteroid, collisions with the player, generation of a new asteroid, and position of a newly created asteroid.

6. Scout: Derived from the enemy class and represents a scouting alien. This alien appears on the screen for a brief time period, which, if shot, gives bonus score to the user. If not, the scout will move off the screen and will reappear in random intervals. Scout will be responsible for the movement of the scout class, speed, interval timer, rendering of the scout, and overall randomization of all of its characteristics.

7. BlasterAliens: Inherits the scouts characteristics in terms of health, interval timer, movement, spawn location, but will stay on-screen until it is killed and will continuously stay at a certain distance away from the player model. It will be responsible for handling the movement bounds (staying within the screen border and a specific distance away from the player), and rendering of the blaster aliens class.

8. ChargingAliens: Inherits the scouts characteristics in terms of health, interval timer, and spawn location, but its movement will be functionally different (e.g., will continuously track the player and accelerate towards them), and will also stay on-screen until it is killed. It will also be responsible for handling the movement bounds and rendering similar to blaster aliens, but for the charging aliens class.

Data and Function Members



https://lucid.app/lucidchart/60b7dbb3-f936-45f3-a2a6-bea5b2fcb40f/edit?viewport_loc=-1740%2C-321%2C3892%2C2053%2C0_0&invitationId=inv_ba43f4b7-16ae-4599-b804-118d1d1d435c

Relationships between Classes

Player has an aggregation relationship with PlayerProj by holding a vector of projectiles, allowing the player to fire multiple projectiles during the game.

Projectiles is the parent class of PlayerProj and EnemyProj.

PlayerProj inherits from Projectiles, allowing projectiles fired by the player to share common behaviours with any future projectile types.

Enemy is the parent class of Asteroids and Scout.

Asteroids inherits from Enemy, which provides basic movement, health, and collision logic, allowing future asteroids to share common behaviours.

Asteroids and Player interact through collision detection, with asteroids having functions to handle collisions with both Player and PlayerProj.

Scout inherits from Enemy, which provides basic movement, health, and collision logic, allowing future scouts to share common behaviours.

BlasterAliens inherit from Scout, which provides basic movement patterns, health, interval timers, and collision logic, allowing future BlasterAliens to share common behaviours.

BlasterAliens has an aggregation relationship with EnemyProj by holding a vector of projectiles, allowing the blasters to fire, at consistent intervals, multiple projectiles at the player during their lifetime.

ChargingAliens inherit from Scout, which provides basic movement patterns, health, interval timers, and collision logic, allowing future ChargingAliens to share common behaviours.

Project Task List and Timeline

1. Design class structures and relationships (1 day) (Jonah De Vizio, Hemanga Hamal, Kerry Cao)
2. Implement the Player class and its functionality (2 days) (Hemanga Hamal)
3. Unit test Player class (1 day) (Jonah De Vizio, Hemanga Hamal, Kerry Cao)
4. Implement the Enemy abstract base class and its functionality (1 day) (Hemanga Hamal, Kerry Cao)
5. Implement the Asteroids class and its functionality (3.5 days) (Hemanga Hamal, Kerry Cao)
6. Unit test Asteroids class (1 day) (Jonah De Vizio, Hemanga Hamal, Kerry Cao)
7. Implement the Projectiles abstract base class and its functionality (1 day) (Hemanga Hamal, Kerry Cao)
8. Implement derived class PlayerProj from Projectiles base class and its functionality (1 day) (Jonah De Vizio, Hemanga Hamal)
9. Unit test PlayerProj class (1 day) (Jonah De Vizio, Hemanga Hamal, Kerry Cao)
10. Implement the Scout class and its functionality (2-3 days) (Kerry Cao)
11. Implement the BlasterAliens class and its functionality (2 days) (Hemanga Hamal)
12. Implement derived class EnemyProj from Projectiles base class and its functionality (2 days) (Jonah De Vizio)
13. Unit test Scout class (1/2 day) (Jonah De Vizio, Hemanga Hamal, Kerry Cao)
14. Unit test BlasterAliens class (1/2 day) (Jonah De Vizio, Hemanga Hamal, Kerry Cao)
15. Implement the ChargingAliens class and its functionality (2 days) (Jonah De Vizio, Hemanga Hamal, Kerry Cao)
16. Create a working UI template for the main game (1 day) (Hemanga Hamal)
17. Create a working version of the main game including all functionality (2 days) (Jonah De Vizio, Hemanga Hamal, Kerry Cao)
18. Debugging and refining the code (1 day) (Jonah De Vizio, Hemanga Hamal, Kerry Cao)

User Interaction Description

The user will interact with the program through using keyboard and mouse controls. The keyboard controls that will be used will be for movement – W,A,S,D – whereas the mouse controls will be used for firing projectiles – Left mouse button – and aiming the player model – tracking of the mouse cursor.

Before the start of the game, the user will have the option to input ‘H’ to open a help menu, discussing the features of the game and how to control the player.

On the game-over screen, the user will be prompted through keyboard inputs to close or restart the game with the letter ‘R’.

Unit Testing and Debugging Plan

The Asteroid game will be tested through both unit testing and organized input/output testing. The unit tests focus on the interaction between each class and the player, which ensures that important game mechanics work seamlessly. In particular, unit testing enables smooth interactions between the player and other classes, like the asteroids class and alien class. For the player class, the unit testing will verify the players movement, firing mechanic, health, and collision detection. Input/Output testing will verify the game correctly reading and writing data from the files and inputs from the user. Error logs will be added, but removed after all errors have been resolved. A MAKEFILE will be included, which will handle the compiling of the files.