[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

[Insert a comprehensive and well-defined list of potential data and function members for each class here that aligns with the class's responsibilities and demonstrates encapsulation. Be sure to include how each data and function member contributes to the class's responsibilities and how it will interact with other classes.]

# 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

[Insert a well-defined and comprehensive plan for unit testing and debugging here. Be sure to include specific testing methods, criteria for success, and how debugging will be handled in the event of errors or issues.]

## Project Title: Video Game Inventory Management System

**Use Case Description**

The Video Game Inventory Management System is a program designed to help video game store owners manage their inventory efficiently. The system will enable store owners to add, delete, and update physical and digital video games, manage customer information, and track transactions. The project's goal is to provide an intuitive and functional user interface that allows store owners to manage their inventory with ease.

## Class List

1. Game: Represents a video game and has attributes like title, developer, publisher, release date, genre, and rating.
2. PhysicalGame: Represents a physical copy of a video game and inherits from the Game class. It has additional attributes like the condition of the disc and its availability status.
3. DigitalGame: Represents a digital copy of a video game and inherits from the Game class. It has additional attributes like the platform it can be played on and the download link.
4. Inventory: Represents the inventory of the video game store and contains all the physical and digital games.
5. Customer: Represents a customer of the video game store and has attributes like name, email, and a unique customer ID.

## Data and Function Members (UML can be used here as well)

* + Game:
    - Attributes:
      * title: string
      * developer: string
      * publisher: string
      * release\_date: date
      * genre: string
      * rating: float
    - Functions:
      * play(): a virtual function that can be overridden by the derived classes to provide different implementations of the function.
  + PhysicalGame:
    - Attributes:
      * condition: string
      * availability\_status: bool
    - Functions:
      * get\_condition(): returns the condition of the disc
      * is\_available(): returns the availability status of the game
  + DigitalGame:
    - Attributes:
      * platform: string
      * download\_link: string
    - Functions:
      * get\_platform(): returns the platform the game can be played on
      * get\_download\_link(): returns the download link of the game
  + Inventory:
    - Attributes:
      * physical\_games: vector of PhysicalGame
      * digital\_games: vector of DigitalGame
    - Functions:
      * add\_game(game): adds a game to the inventory
      * remove\_game(game): removes a game from the inventory
      * update\_game(game): updates the details of a game in the inventory
      * search\_game(title): searches for a game by its title in the inventory
      * display\_inventory(): displays the inventory of physical and digital games
  + Customer:
  + Attributes:
    - name: string
    - email: string
    - customer\_id: int
  + Functions:
    - get\_name(): returns the name of the customer
    - get\_email(): returns the email of the customer
    - get\_customer\_id(): returns the customer ID of the customer

## Relationships between Classes (UML can be used here as well)

* Game is the parent class of PhysicalGame and DigitalGame.
* PhysicalGame and DigitalGame inherit the attributes and functions of the Game class.
* The Inventory class contains both physical and digital games.
* The Customer class is not related to any other class in the system.

## Project Task List and Timeline

1. Design the class structure and relationships (x days) (group member x)
2. Implement the Game, PhysicalGame, and DigitalGame classes (x days) (group member x)
3. Implement the Inventory and Customer classes (x days) (group member x)
4. Implement the user menu and user interface (x days) (group member x)
5. Implement the file input/output functions (x days) (group member z,y)
6. Write unit tests for all classes and functions (x days) (group member x,y)
7. Debug and refine the code (x days) (group member x,y,z)
8. Write user documentation (x days) (group member x,y,z)

## User Interaction Description

The user will interact with the program through a user-friendly menu that provides options for adding, deleting, and updating games in the inventory, managing customer information, and viewing transaction history. The menu will be implemented using prompts that will allow the user to select an option by entering a corresponding number or character.

The program will also use error messages and confirmation prompts to guide the user through the program and ensure that the input is valid. For example, if the user tries to add a game that already exists in the inventory, the program will display an error message and prompt the user to enter a different game.

The program will allow the user to view the inventory of physical and digital games by selecting the corresponding option from the menu. The user can search for a game by its title and view its details, including its availability status, condition, platform, and download link.

The user can also add and manage customer information by selecting the customer management option from the menu. The program will allow the user to add a new customer, update customer information, and view a list of all customers in the system.

## Unit Testing and Debugging Plan

The program will be tested using a combination of unit testing and organized input/output testing. The unit tests will cover each function of the classes and will ensure that the program performs as expected. The input/output tests will verify that the program reads and writes data correctly from the file and that the program handles user input and output correctly.

The program will also use exception handling to handle unexpected inputs or situations and prevent crashes or errors. The program will log any errors or issues encountered during runtime and will provide error messages or prompts to guide the user through the program.

Makefile and README files will be included in the project, which will provide clear instructions on how to compile, run and test the program. The Makefile will handle dependencies, and the debugging and release builds will be separated. The README file will explain how to use the program and will list any known issues or limitations of the program.