	UNIVERSITI TEKNOLOGI MALAYSIA

Programming Technique II
(SECJ 1023)

Project Deliverable 2

Project Proposal
Problem Analysis and Design

Student Name	**: 1. Evelyn Ang (A24CS0068)**
	2. Angela Ngu Xin Yi (A24CS0226)
	3. Tan Xin Tian (A24CS0198)
	4. Toh Shee Thong (A24CS0309)
Lecturer Name	**: Dr Shamini A/P Raja Kumaran**
Section	**: 2**

Table of Content

1.0 Summary

In recent years, 2D shooting games have continued to captivate players with their dynamic, fast-paced gameplay and straightforward mechanics. Our project, Sky Defender, is a 2D aerial shooting game. It is inspired by the classic arcade title 1945 Air Force. It retains the nostalgic feel of classic arcade shooters while adding updated features to improve the gaming experience.

Sky Defender features a top-down view where players control a fighter aircraft tasked with surviving relentless waves of enemy planes. The gameplay progressively increases in difficulty. This requires players to rely on quick reflexes, strategic movement and precise shooting. Inspired by the 1945 Air Force's addictive arcade formula, Sky Defender aims to deliver an updated version with enhanced graphics, more immersive missions, customizable aircraft and tactical decision-making elements.

In addition to simple shooting mechanics, Sky Defender also incorporates elements of aerial combat simulation. Players will experience different mission types such as air-to-air engagements and precision bombing runs. At the same time, they can manage realistic aircraft handling and diverse weapon systems. Sky Defender promises to provide an engaging and immersive experience, regardless of whether a casual gamer seeking explosive arcade fun or a hardcore flight sim fan looking for tactical complexity.

2.0 Problems/Weaknesses

The limited mission variety is one of the key weaknesses identified in traditional 2D airplane shooting games. Many of these classic games tend to offer repetitive gameplay. Players have to perform the same basic actions across multiple levels with little variation in the goals or difficulties. Over time, players who are looking for fresh and interesting experiences as they go may eventually become weary of this repetitive framework and lose interest.

Another common shortcoming is the lack of customization options. In many older 2D shooters, players are restricted to using a predetermined set of aircraft. Players are unable to personalize their appearance, abilities or weaponry. This limitation reduces the sense of ownership and connection players have with their in-game avatar. Ultimately, this decreases replayability.

Furthermore, most of the conventional 2D shooters have outdated sound and visual designs even though retro graphics and classic soundtracks have a nostalgic appeal. Modern players might not be satisfied by the repeated sound effects, simple animations and static backgrounds. For this reason, players nowadays are used to more dynamic and engaging gaming environments. These factors combined highlight the need for improvements in mission design, player customization and aesthetic presentation to create a more engaging and enjoyable gaming experience for today's audience.

3.0 Game Mission

In Sky Defender, the player controls a fighter aircraft that can move freely across the screen using the arrow keys (↑ ↓ ← →). This allows the player to navigate in all directions. It provides users the flexibility needed to dodge incoming enemies and position themselves strategically.

The fighter plane is equipped with an automatic firing system. The plane can continuously shoot bullets upwards at regular intervals without the need for manual input. This ensures smooth and fast-paced gameplay, keeping the player's focus on movement and survival.

The scoring system rewards the player with 10 points for each enemy aircraft successfully destroyed. The current score is prominently displayed on the screen throughout gameplay to offer real-time feedback and encourage players to beat their previous records.

The game also features a life system, where the player starts with three lives. Each collision with an enemy results in the loss of one life. When all three lives are depleted, the game ends. In the end, the final score is displayed. This can provide a clear conclusion to the session and motivate players to improve in subsequent attempts. These core mechanics together create an accessible yet challenging experience that balances action, strategy and skill.

4.0 Storyboards

4.1 Scene 1

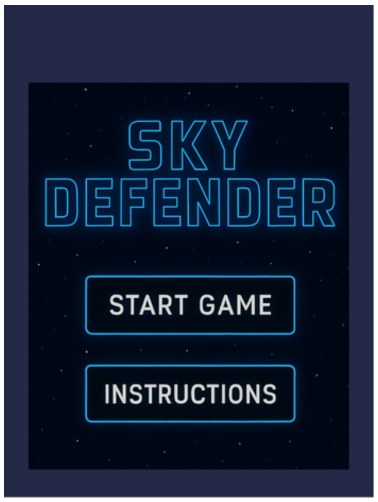


Figure 1: Home Page

Player enters the game and clicks “Start Game” to begin the mission. The instructions will be shown when the player clicks “Instructions”.

4.2 Scene 2

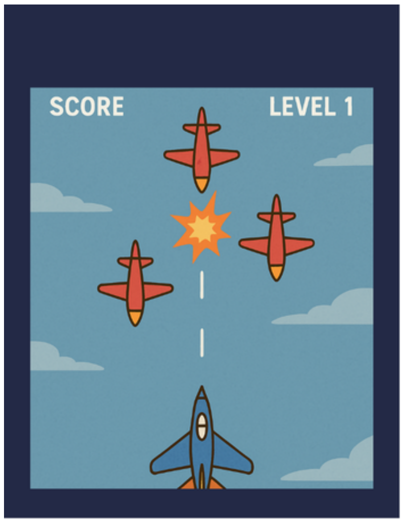


Figure 2: Game Interface

Enemies are flying down from the top. The plane would automatically shoot the bullets. Player will score 10 points for each when they shoot down the enemies.

4.3 Scene 3

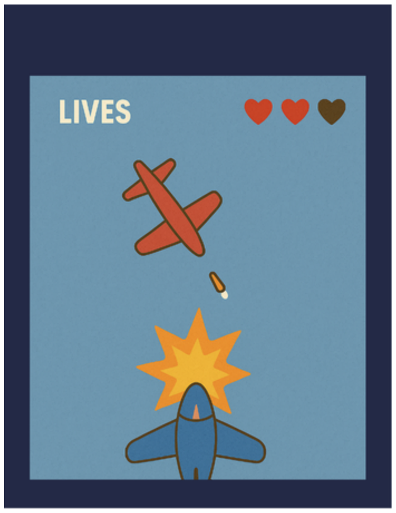


Figure 3: Player’s jet hit by a bullet

If the player collides with an enemy plane or gets hit by a bullet, 1 life will be lost.

4.4 Scene 4

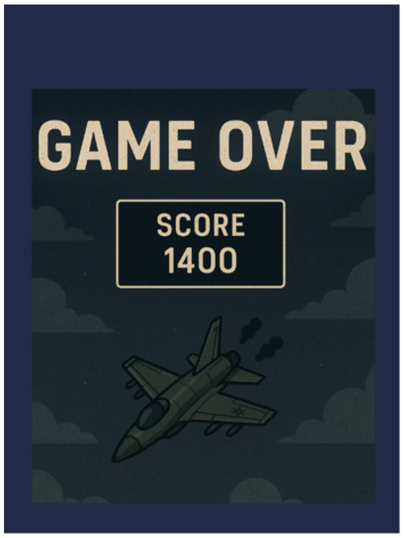


Figure 3: Game Over

When all lives are lost, the game ends! Finally, the game shows the final score.

5.0 Benefits of our projects

One of the main advantages in Sky Defender is the auto-shooting function which eliminates the necessity for manual fire inputs. In many shooting games, players need to press or click a button continuously such as a spacebar in order to shoot, which might be intimidating for the beginners. Therefore, Sky Defender simplifies this process such that the bullets are automatically fired at controlled intervals. This allows players to only focus on the movement and the strategy. This design makes the game more accessible because the new player can quickly adapt without frustration while still enjoying fast-paced action.

Additionally, Sky Defender creates the enemies that move at different speeds so that the game is always dynamic and unpredictable. Unlike traditional shooters in which the enemies move in the fixed speeds that are easy to anticipate, this game randomizes their velocity. As a result, the player must adjust their tactics and respond quickly. Some enemies move slowly and need cautious movement, while others rush at the player and demand an instant response. In addition to making each gameplay experience distinct, this variation helps players improve their response speeds.

Furthermore, Sky Defender provides a dynamic moving background in order to create a visually stimulating environment which is essential for an immersive gaming experience. The starfield is moved continuously, giving the game world depth and simulating fast flight. This little but impactful element adds a sense of motion which makes the gaming more engaging than static backdrops. This changing background, when combined with the auto-shooting and randomly generated opponent speeds, makes Sky Defender feel contemporary while adhering to its arcade origins.

6.0 Classes

In this project, the following classes are designed to structure the game development:

6.1 GameObject

The GameObject marks the beginning of all visual representations of objects within the game. No direct instances of GameObject are ever created at the time of gameplay, but it establishes fundamental properties and behaviors that all game objects inherit. It includes attributes such as x and y coordinates to define the object's position, width and height for collision detection, and color for rendering purposes. To access these attributes, the class offers the getX(), getY(), getWidth(), getHeight(), and getColor() methods. Additionally, it works with collision detection through the isColliding(const GameObject* other) method which finds object overlaps using the Axis-Aligned Bounding Box (AABB) algorithm. GameObject marks two pure virtual functions—update() for object logic and draw() for rendering. Doing so guarantees that the subclass design will be changed in a controlled manner. This abstract parent enables uniform interaction patterns while maintaining individual game objects with their specific properties and behaviors within the shared game world.

6.2 Player

The Player class, derived from GameObject, represents the player-controlled jet in the game. It includes attributes such as lives, which monitor health, score to earn points, and shootCooldown, a timer that controls firing speed to prevent rapid shooting. The class offers movement methods (moveLeft(), moveRight(), moveUp(), moveDown()) to allow the player to move around the screen. In the scope of game logic, addScore() is implemented for incremental point increases and loseLife() reduces the health parameter if the user takes damage. Furthermore, canShoot() determines whether bullets can be fired for the period of no fire is after the cooldown period. The class overrides update() to handle cooldown management and other real-time logic, while draw() renders the aircraft as a colored triangle.

6.3 Bullet

The Bullet class, derived from GameObject, represents projectiles fired by both the player and enemy jets. It inherits core attributes such as position (x, y), size (width, height) and color. Furthermore, it also introduces a boolean flag isEnemyBullet to differentiate between bullet types. During construction, the bullet's position and color are initialized based on whether it belongs to the player or an enemy. Player bullets start slightly above the player aircraft and move upward, while enemy bullets spawn below the enemy and move downward.

The overridden update() method adjusts the vertical position of the bullet based on its type. For example, the position is adjusted upward for player bullets and downward for enemy bullets. The draw() method renders the bullet as a simple colored rectangle on the screens. The isOutOfBounds() method checks if the bullet has left the screen which is top for player bullets and bottom for enemy bullet. The function flags it for deletion to maintain performance and prevent memory overflow. This class also plays a crucial role in gameplay mechanics such as collision detection, score incrementing, and life reduction depending on whether the bullet hits an enemy or the player.

6.4 Enemy

The Enemy class, derived from GameObject, represents hostile aircraft that descend from the top of the screen toward the player's ship. In addition to inherited attributes, it includes a speed value, which is randomly initialized to add variation in enemy behavior. Other than that, a shootCooldown attribute regulates the firing interval for enemy bullets.

Each enemy maintains an array of Bullet pointers (bullets[]), enabling it to fire projectiles downward. The tryShoot() method handles bullet creation with a cooldown timer, returning a new Bullet object when shooting is allowed. The overridden update() method moves the enemy downward each frame. The draw() method renders it as a simple inverted triangle, visually distinguishing it from the player. The isOutOfBounds() method ensures that enemies exiting the screen's lower boundary are removed to maintain performance. This class supports dynamic enemy behavior, contributing to the game's difficulty and variety through randomized speed and shooting patterns.

6.5 Game

The Game class serves as the central management system. The primary operations are responsible for input processing, updating game states, rendering graphics and handling collisions. It maintains important game elements through several key attributes: a player pointer representing the user-controlled aircraft; appearing and disappearing bullets and enemies stored as vectors of pointers; a gameOver flag to track termination conditions; an enemySpawnTimer to regulate enemy generation frequency; and backgroundY for setting a dynamic scrolling background effect.

The class implements fundamental methods to ensure proper game execution: The processInput() method interprets user commands for aircraft navigation and firing actions. The update() method is responsible for advancing the state of the game world by moving objects, checking for collisions between game elements and managing spawn timers. Visual representation is handled through the draw() method, which renders all active game objects while displaying relevant UI elements such as score and health bars. Memory management is taken care of via the cleanup() method, which systematically removes dynamically allocated resources to prevent memory leaks. Finally, the run() method glues all the systems together into a game loop that ensures the game is processed, updated, and rendered in real-time. This architecture is flexible enough, supporting easy addition of new features.

7.0 Object

In this project, several key objects have been identified based on the game requirements and designs. Each object plays an important role in the game. The objects are described below:

7.1 Player

The Player object serves as the representation of the user's aircraft within the game world. This aircraft can be controlled using keyboard inputs to move in four directions (left, right, up, and down) and is maintained within the boundaries of the game window. The Player object will keep firing bullets at regular time intervals without requiring manual input for each shot, thus creating a steady stream of bullets. It maintains important game state information, such as the player's remaining lives, which are initially three and decrease upon hitting enemies, and a cumulative score that increases when bullets successfully hit enemy targets. The Player object will interact with other game objects, updating itself on collision and performing user-initiated commands for moving around within the game space.

7.2 Bullet

The Bullet object represents the projectiles fired by both the player's aircraft and enemy aircraft during gameplay. Each bullet maintains its own position, size and movement direction. Upon creation, its behavior is determined by a boolean flag. If the bullet is fired by the player, it spawns just above the player's aircraft and travels upward. By the same token, if fired by an enemy, it spawns just below the enemy and travels downward. The color and initial position of the bullet are set differently based on who fired it, providing visual clarity during gameplay. The bullets continuously update their position in each game frame and are removed from memory when they travel beyond the screen bounds. Player bullets, when colliding with enemies, contribute to the player's score and are then deleted, whereas enemy bullets pose a threat to the player's aircraft. This dual behavior supports dynamic and engaging gameplay, requiring players to dodge enemy fire while aiming their own shots precisely.

7.3 Enemy

The Enemy object represents enemy aircraft that enter the gameplay area from various positions along the top of the screen. Visually rendered as inverted triangle shapes, these enemies are created by the Game at regular intervals of time. These enemy aircraft will descend towards the player at various speeds, with some moving quite rapidly and others slowing them down in an attempt to provide variety in the challenge of the game. Moreover, they can shoot bullets toward the player. The enemy objects are removed from play under two conditions: when they successfully reach the bottom screen boundary or when they collide with the player's plane or the player's bullets. Collisions between the player's aircraft and enemies subtract from the life count of the player, whereas collisions involving the player's bullets do not affect the player's aircraft but reward the player points while removing the enemy from play.

7.4 Game

The Game object is the central management system that manages everything regarding the gameplay. This top-level controller maintains references to all active game objects, such as the player, active bullets, and any existing enemies. The Game object captures and interprets player input, translates it into movement or actions like shooting, and updates the state of every active object on the screen. It also handles essential tasks such as background scrolling, score tracking, enemy spawning at timed intervals, and complex collision detection logic between bullets, enemies, and the player. Lastly, the Game object is responsible for rendering all visual elements to the screen using double buffering to ensure smooth animations. By integrating these systems, the Game object ensures a cohesive and interactive gameplay loop that functions reliably and responsively.

8.0 Relationship

8.1 Association

Association is a relationship between two classes where one class uses or is connected to another class without inheriting from it. One class holds a reference to objects of another class to perform specific tasks while both classes remain independent. Association allows different classes to work together without tightly coupling their behaviors, maintaining modularity and flexibility in the program design.

Class: Game

Associated Class	Explanation
Player	The Game has a Player. The Game class creates and owns a single Player object (Player* player), representing the user-controlled aircraft. It handles the player's input for movement and shooting, manages the player's score and lives, and draws the player on the screen. The Game class also checks for collisions between the player and enemy bullets or enemies to update the player's lives and game state. This is a strong "has-a" association where the Game fully manages the lifecycle of the Player object (created in the constructor and deleted in cleanup()).
Bullet	The Game has many Bullets, consisting of both player-fired bullets and enemy-fired bullets. For player bullets, the Game class does not directly manage them. Instead, it accesses and interacts with the player's bullets through the Player object using methods like getBullets() and getBulletCount(). The Game interacts with bullets without owning them. In contrast, the Game class directly creates and manages enemy bullets using an array (Bullet* enemyBullets[MAX_ENEMY_BULLETS]). When enemies shoot, the resulting bullets are stored and updated in this array. The Game class is responsible for updating their positions, checking for collisions with the player and deleting them when they move off-screen or after impact.
Enemy	The Game has many Enemies. The Game class controls the spawning, updating, and deletion of Enemy objects using an array (Enemy* enemies[MAX_ENEMIES]). It determines when new enemies appear, updates their positions, handles their shooting via tryShoot(). Then, it removes them when they go out of bounds or are destroyed by player bullets or collisions.

8.2 Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP) where one class (the child) inherits properties and behaviors (attributes and methods) from another class (the parent). It is used since child classes can modify or extend the behavior of parents to meet specific requirements and can be used to represent a hierarchical relationship between classes, which helps organize code logically.

Parent Class: GameObject

Child Class	Explanation
Player	Player is a GameObject. It inherits properties like position (x, y), size (width, height) and color from GameObject. It specializes the GameObject by adding player-specific behaviors like moving left, right, up, and down within the screen boundaries, handling shooting cooldowns, tracking the player's score and lives, and customizing how the player's aircraft is drawn as a triangular shape.
Bullet	Bullet is a GameObject. It inherits from GameObject and overrides draw() and update() to represent upward movement and display a bullet shape. It represents a bullet fired by the player, moving vertically upwards by decreasing its y-coordinate over time. It also has a simple rectangular visual shape drawn using a bar.
Enemy	Enemy is a GameObject. It inherits properties like position (x, y), size (width, height) and color from GameObject. It customizes draw() and update() to simulate enemy aircraft behavior. In update(), enemies descend vertically at varying random speeds to make the gameplay more dynamic and unpredictable. In draw(), enemies are visually represented as inverted triangles, differing from player and bullet. The class also introduces additional functionality such as shooting with a cooldown mechanism, where enemies can only fire bullets after a certain delay (shootCooldown), adding further complexity and realism to their behavior.

8.3 Composition

Player class HAS A Bullet class

Player class has Bullet class relationship which Player class contains an array of Bullet pointers (Bullet* bullets[MAX_PLAYER_BULLETS];). The player is solely responsible for creating (new Bullet(...)) and destroying (delete bullets[i]) these bullets. When a Player object is destroyed, it cleans up its owned Bullet objects in its own destructor.

8.4 Aggregation

Enemy HAS Bullet (but does not manage them)

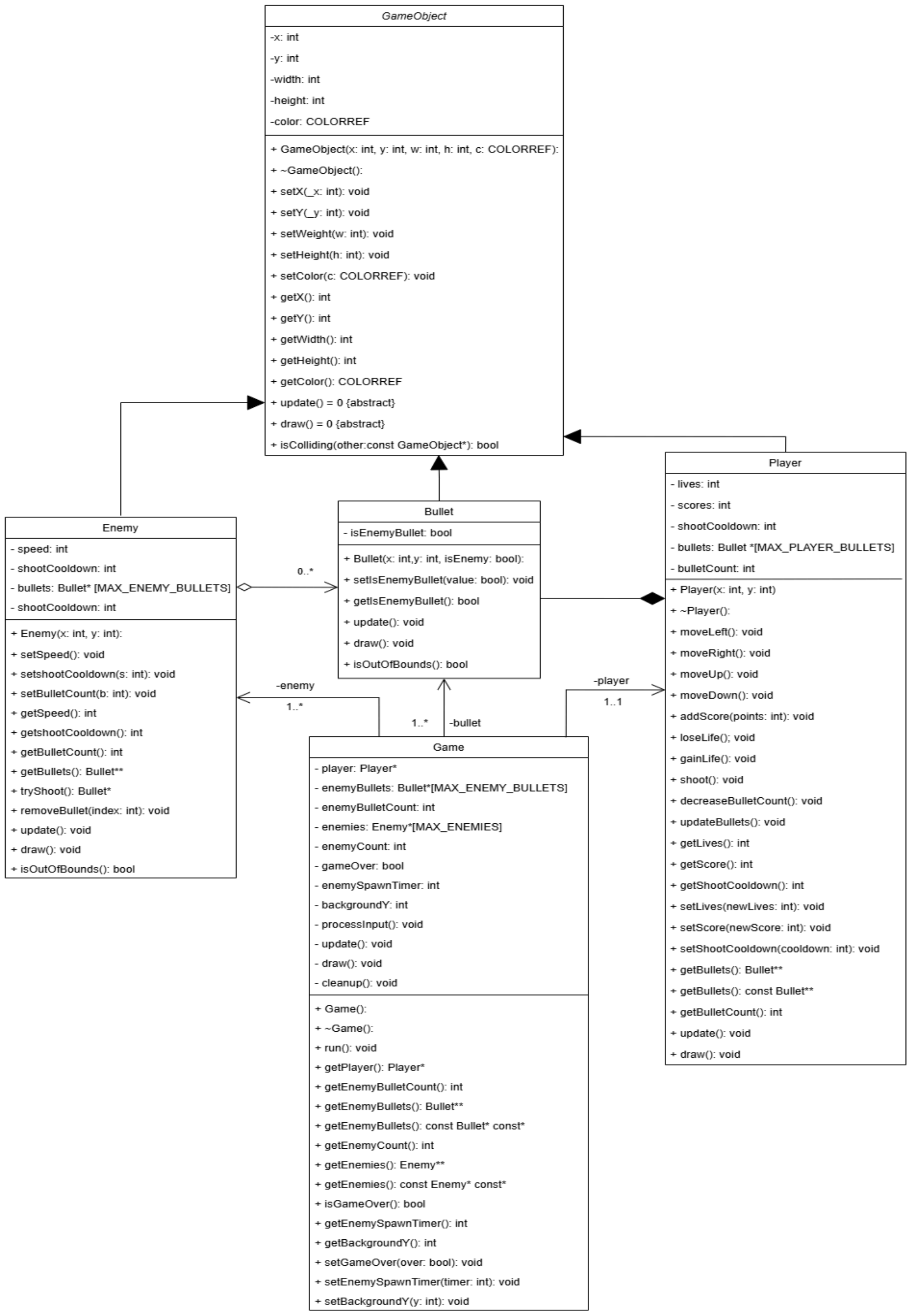
In aggregation, an object can use another object but does not own or manage its entire lifecycle. This can be observed in the relationship between Enemy class and Bullet class. The Enemy class can shoot bullets using tryShoot() method. It shows that the Enemy created the bullets but doesn’t store it. Game receives and manages the bullet. When Enemy class is destroyed, its bullets continue to exist in Game class. Enemy class creates but does not own its bullets. The bullets live independently of the Enemy class.

8.5 Polymorphism

GameObject base class enables polymorphic behavior

Polymorphism is the relationship that allows objects of different classes to be treated through a common interface. This is implemented through GameObject class. GameObject::update() and GameObject::draw() are pure virtual functions, making GameObject an abstract base class. This forces all concrete derived classes which is Bullet, Player and Enemy class to provide their own implementations for how they update their state and draw themselves on the screen.

9.0 UML Class Diagram


GameObject
-x: int
-y: int
-width: int
-height: int
-color: COLORREF
+ GameObject(x: int, y: int, w: int, h: int, c: COLORREF):
+ ~GameObject():
+ setX(_x: int): void
+ setY(_y: int): void
+ setWeight(w: int): void
+ setHeight(h: int): void
+ setColor(c: COLORREF): void
+ getX(): int
+ getY(): int
+ getWidth(): int
+ getHeight(): int
+ getColor(): COLORREF
+ update() = 0 {abstract}
+ draw() = 0 {abstract}
+ isColliding(other:const GameObject*): bool
Enemy
- speed: int
- shootCooldown: int
- bullets: Bullet* [MAX_ENEMY_BULLETS]
- shootCooldown: int
+ Enemy(x: int, y: int):
+ setSpeed(): void
+ setshootCooldown(s: int): void
+ setBulletCount(b: int): void
+ getSpeed(): int
+ getshootCooldown(): int
+ getBulletCount(): int
+ getBullets(): Bullet**
+ tryShoot(): Bullet*
+ removeBullet(index: int): void
+ update(): void
+ draw(): void
+ isOutOfBounds(): bool
0..*
Bullet
- isEnemyBullet: bool
+ Bullet(x: int,y: int, isEnemy: bool):
+ setIsEnemyBullet(value: bool): void
+ getIsEnemyBullet(): bool
+ update(): void
+ draw(): void
+ isOutOfBounds(): bool
Player
- lives: int
- scores: int
- shootCooldown: int
- bullets: Bullet *[MAX_PLAYER_BULLETS]
- bulletCount: int
+ Player(x: int, y: int)
+ ~Player():
+ moveLeft(): void
+ moveRight(): void
+ moveUp(): void
+ moveDown(): void
+ addScore(points: int): void
+ loseLife(); void
+ gainLife(): void
+ shoot(): void
+ decreaseBulletCount(): void
+ updateBullets(): void
+ getLives(): int
+ getScore(): int
+ getShootCooldown(): int
+ setLives(newLives: int): void
+ setScore(newScore: int): void
+ setShootCooldown(cooldown: int): void
+ getBullets(): Bullet**
+ getBullets(): const Bullet**
+ getBulletCount(): int
+ update(): void
+ draw(): void
-enemy
1..*
1..*
-bullet
-player
1..1
Game
- player: Player*
- enemyBullets: Bullet*[MAX_ENEMY_BULLETS]
- enemyBulletCount: int
- enemies: Enemy*[MAX_ENEMIES]
- enemyCount: int
- gameOver: bool
- enemySpawnTimer: int
- backgroundY: int
- processInput(): void
- update(): void
- draw(): void
- cleanup(): void
+ Game():
+ ~Game():
+ run(): void
+ getPlayer(): Player*
+ getEnemyBulletCount(): int
+ getEnemyBullets(): Bullet**
+ getEnemyBullets(): const Bullet* const*
+ getEnemyCount(): int
+ getEnemies(): Enemy**
+ getEnemies(): const Enemy* const*
+ isGameOver(): bool
+ getEnemySpawnTimer(): int
+ getBackgroundY(): int
+ setGameOver(over: bool): void
+ setEnemySpawnTimer(timer: int): void
+ setBackgroundY(y: int): void