Ray Vanguard Design Report

Introduction

For this program, I want to build a classic arcade game with a roguelike concept in it. Enemies are coming from the top of the screen. Players can shoot bullets to destroy the enemy ship. If a player touches the enemy, the game is over. Players can upgrade after destroying a set amount of enemies. The game will get harder and harder every time, including but not limited to, enemies spawn, how many to destroy until the next stage, etc. There is no ending in this game, player are going for their best marks.

Figure 1 - Game Demo
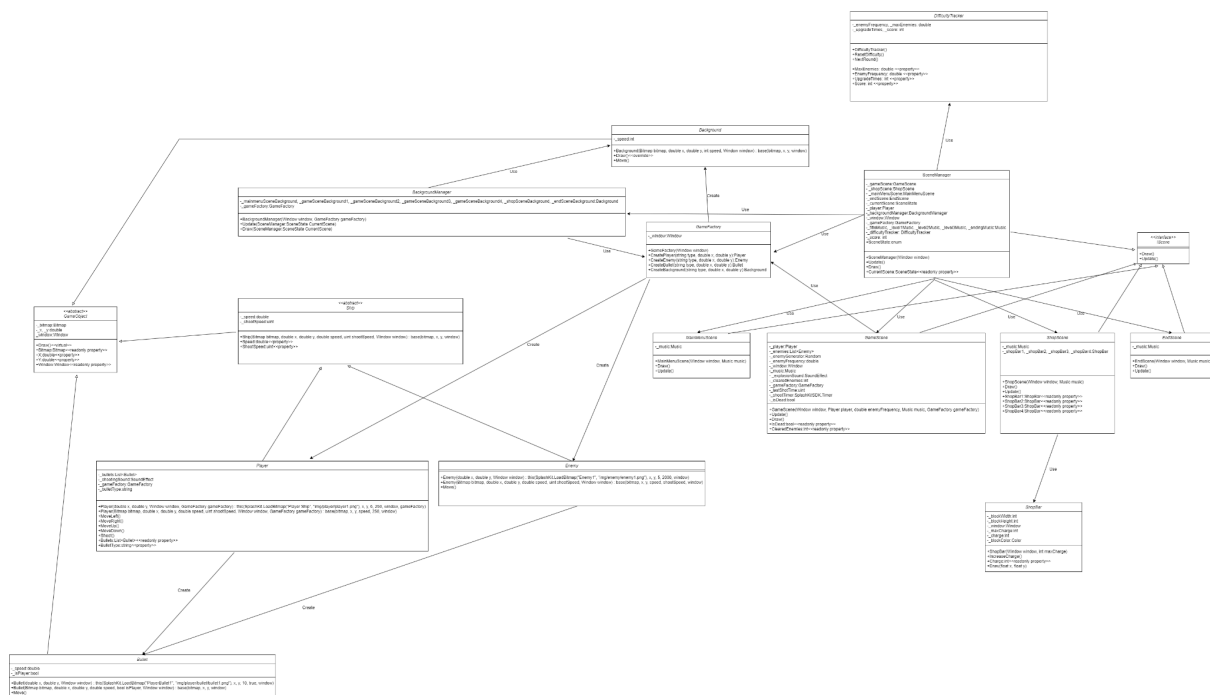
UML of the program

Figure 2 - UML

The UML has been slightly changed after the original program design. I will go through briefly how each class should behave, and why are they needed.

Class Table

Class name	What does it do	Why is this needed
Background	Child of GameObject, responsible for drawing the background	Offers the game object to be able to move and loop itself
BackgroundManager	Responsible for managing all the background	Able to create and manage multiple backgrounds. Also useful when scene changes
Bullet	Child of GameObject, Bullet shot by player	Able to move and detect what the bullet

	and enemy	belongs to. Also great for collision detection.
DifficultyTracker	Manage the difficulty of different stages	To manage the difficult changes. Having a dedicated class for this is better for code readability.
EndScene	Use the IScene interface, this class is responsible for the ending scene that shows after the player gets hit.	Represent the ending scene.
Enemy	Child of Ship class. Responsible for enemies.	Offers an enemy and allows different enemies to be spawned with different passes in value.
GameFactory	For spawning different object prefabs.	Lower the classes needed for creating multiple game object types.
GameObject	Abstract class, every object that is supposed to be drawn on the screen using bitmap should be a child of GameObject.	Ensures that code is minimized by having some methods(draw) in the parent class.
GameScene	Use the IScene interface, this class is responsible for the game scene that shows when a player is playing.	Represent the game scene
IScene	An interface, which every scene should use.	Make sure that every scene has the basic functions such as

		Draw and Update.
MainMenuScene	Use the IScene interface, this class is responsible for the main menu scene.	Represent the main menu scene.
Player	Child of the ship, Responsible for the player ship.	Represent the character the player use.
Program	Program	Program
SceneManager	Child of IScene manages all of the scenes	Able to create and manage all of the scenes, also manage scene change
Ship	Abstract class, every movable ship in this game should be a child of this class.	Ensures that code is minimized by having some methods(move) in the parent class.
ShopBar	Shopbar to indicate the upgrade. Draw with build-in SplashKit method, thus not belonging to GameObject.	Keeping track of how many upgrades the user has.
ShopScene	Use the IScene interface, this class is responsible for the game scene that shows when a player enters the shop	Represent the shop scene.

One of the main methods or design features

One of the main methods or design features is definitely the scene change. By having an enum that stores the scene, and a switch case to change the behavior. I am able to have different scenes that behave differently. This opens up a lot of potential, one of them being the shop scene, which happens after players kill a set amount of enemies. Players can upgrade in the shop, and then get back into the game scene with a more dangerous stage.

Use of Abstraction

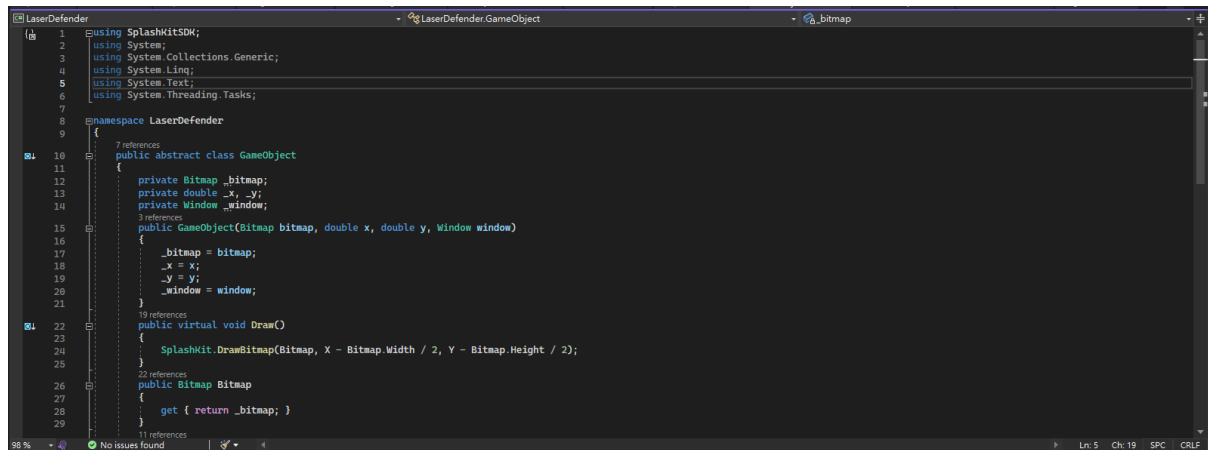Abstraction is everywhere in this program.

Figure 3 - Code demo

GameObject is an abstract class. Because every bitmap needs a draw method to appear on the screen. And since the original draw method draws from the top left corner, which is not good for collision detection. So it's better to have a GameObject class to manage it.
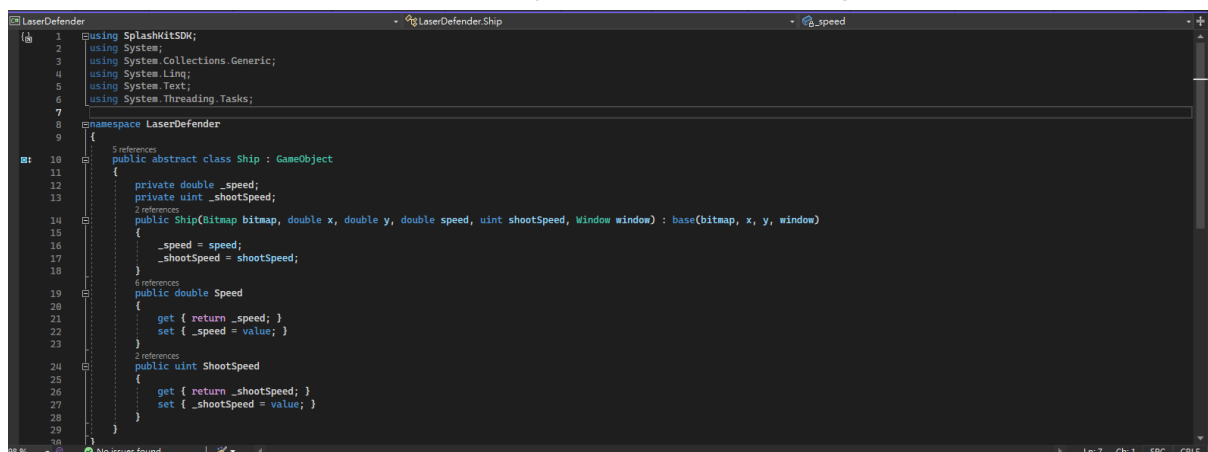

Figure 4 - Code demo

The ship is also an abstract class. Because all of the classes should have speed and shoot speed pass in. By having the ship as the parent class we can ensure that those are in the class while not duplicating code.

The use of inheritance and polymorphism

Inheritance and polymorphism are everywhere. Player and Enemy inherited from Ship class, while Player and Enemy behave differently. Players have a controller that allows Player movement to be controlled. While Enemy has a set amount of movement that can't be changed. But they can both be managed as Ship.
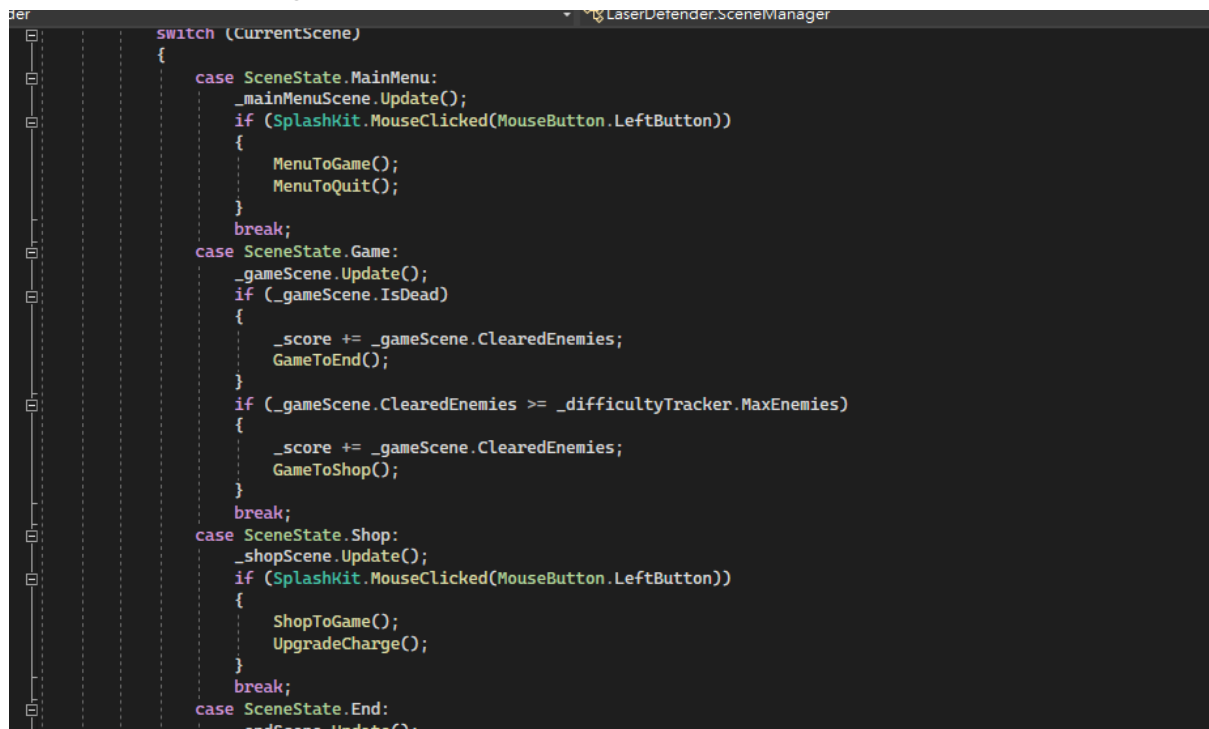
Every single drawing using bitmap is a child of GameObject. Which offers the ability to draw to the screen while not having to duplicate the code for every class.

IScene is an interface that requires all of the scenes to have Draw and Update.

There are many more examples across the program which I will also explain in the interview.
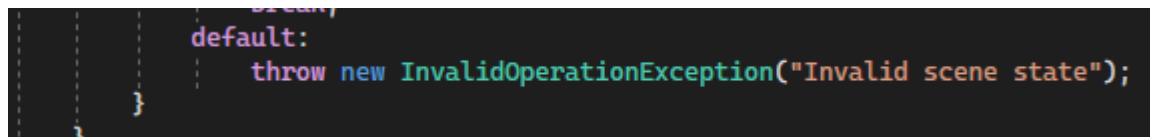
Appropriate use of C# coding conventions

There are quite a bit of examples this one. For example, I've used the scene state changes in a switch case.

```csharp
                                                    ▼ ⚙ LaserDefender.SceneManager
    switch (CurrentScene)
    {
        case SceneState.MainMenu:
            _mainMenuScene.Update();
            if (SplashKit.MouseClicked(MouseButton.LeftButton))
            {
                MenuToGame();
                MenuToQuit();
            }
            break;
        case SceneState.Game:
            _gameScene.Update();
            if (_gameScene.IsDead)
            {
                _score += _gameScene.ClearedEnemies;
                GameToEnd();
            }
            if (_gameScene.ClearedEnemies >= _difficultyTracker.MaxEnemies)
            {
                _score += _gameScene.ClearedEnemies;
                GameToShop();
            }
            break;
        case SceneState.Shop:
            _shopScene.Update();
            if (SplashKit.MouseClicked(MouseButton.LeftButton))
            {
                ShopToGame();
                UpgradeCharge();
            }
            break;
        case SceneState.End:
            endScene.Update();
```

Figure 5 - Code demo

```csharp
        default:
            throw new InvalidOperationException("Invalid scene state");
    }
}
```

Figure 6 - Code demo

I've also set it to throw in a warning if the scene state is not any of the scene states we have right now.

Appropriate use of structured programming principle

I think I've made the structured programming principle very well in my game. I will use the code in Figures 4 and 5 as a demo.

The code uses structured control flow constructs like if statements and switch statements to make decisions and control the flow of the program.

For example, the Update() method, checks the CurrentScene using a switch statement to execute the appropriate logic for each scene.

The code avoids the use of "goto" or other unstructured control flow mechanisms, which would make the code harder to read and maintain.

It follows a clear and readable structure, with each scene's update and logic neatly organized within its respective case statement in the switch block.

The code maintains a structured separation of concerns by dividing functionality into different methods and classes, enhancing code modularity and readability

Pattern used

State Pattern: Scene State
Factory Pattern: GameFactory
Singleton Pattern: GameFactory, BackgroundManager, Window, Music, etc.
Strategy Pattern: Scene Changes at runtime and behavior changes

Difficulty in understanding the code

One thing that I am proud of is how easy to understand the program is. With the proper name for everything, I am able to minimize the comment needed in the program. The player can shoot with the shooting method, SceneManager manages the scene, DifficultyTracker tracks the difficulty, etc. Move() controls the movement, and CreateEnemy() creates an enemy, etc.

For the more difficult parts of the code, I've made different comments to make sure it's easy to understand.

Figure 7 - Code demo