Side Scrolling Shooter

1. Project Overview

This project is a side-scrolling shooter game where players control a character navigating through different levels while eliminating enemies, avoiding obstacles, and collecting power-ups. The game will include smooth movement mechanics, responsive shooting controls, dynamic enemy behavior, and a scoring system. The main objective is to progress through levels while maximizing the score and surviving enemy attacks.

2. Project Review

For the foundation of this project, I will examine an existing 2D side scrolling shooter game built using Python and Pygame. A common implementation includes the basic gameplay where the player controls a character at rows of invaders. However, the original version lacks advanced features such as difficulty progression, power-ups, or tracking statistics. My project will improve upon this by:

- **Adding Level Progression:** Enemies will increase in speed and complexity as the player advances through levels.
- **Power-ups:** Special power-ups item will occasionally drop from defeated invaders, giving the player temporary advantages such as healing or ammos.
- **Score Tracking:** A dynamic scoring system that adjusts based on the player's actions (e.g., number of enemies defeated, accuracy).
- **Enhanced AI:** Instead of just moving left to right, the enemies will vary their behavior by creating different movement patterns (random, etc.).

Programming Development

3.1 Game Concept

The **Side-Scrolling Shooter** is an action-packed 2D game where players control a heavily armed character navigating through enemy-infested environments. The game challenges players to defeat waves of enemies, avoid obstacles, and collect power-ups while progressing through levels of increasing difficulty.

Core Mechanics :

- **Player Movement**: The character moves left or right, jumps over obstacles, and crouches to dodge attacks.
- **Shooting System**: The player can fire in multiple directions and has different weapons with limited ammunition.
- **Enemy AI**: Various enemy types with different attack patterns, such as ranged shooters, melee attackers, and flying drones.
- **Health System**: The player has a limited health pool and can pick up health packs to restore lost HP.
- **Power-ups & Upgrades**: Players can collect special items, such as weapon upgrades, speed boosts, and temporary invincibility.
- **Scoring System**: Players earn points by defeating enemies, collecting power-ups, and completing levels efficiently.

3.2 Object-Oriented Programming Implementation

The game will utilize OOP principles, including the following key classes:

- **Game:** The `Game` class serves as the main engine that initializes and runs the game. It sets up the game window, handles the game loop, and manages the overall flow of the game (e.g., drawing objects and handling events).

  Key Attributes:

- `screen`: A Pygame surface object where all game objects are drawn.
- `running`: A boolean flag to control the game loop's execution.
- `player`: An instance of the `Soldier` class representing the main character.
- `enemy_group`: A group containing enemy `Soldier` instances.
- `bullet_group`, `grenade_group`, `explosion_group`: Groups for projectiles and explosions.
- `item_box_group`: Manages collectible items such as ammo, health, and grenades.
- `decoration_group:`Handles non-interactive decorative objects such as trees, rocks, or background elements. These objects enhance the environment visually but do not affect gameplay.
- `water_group:`Represents water areas in the game, which might slow down the player or be impassable.
- `exit_group:`Contains exit points that trigger level completion when the player reaches them.

- self.moving_left, self.moving_right: Determines if the player is moving left or right.
- self.shoot: Tracks whether the player is shooting.
- self.grenade: Indicates if the player has activated a grenade.
  self.grenade_thrown: Prevents continuous grenade throws (ensuring a single throw per action).
- self.start_game: Likely used to start the game when set to True.
- `clock`: Controls the frame rate (FPS) to ensure smooth gameplay.

  Methods:

- `__init__(self)`: Initializes the game environment, including Pygame, the game screen, and the player objects.
- `draw_text(self, text, font, text_col, x, y)`: Displays text on the screen.
- `draw_bg(self)`: Renders the game background and scene elements.
- reset_level(self): Resets the current level to its initial state. This may include repositioning players, resetting health or stats, clearing enemies or objects, and reloading the map.
- show_stats_screen(self): Displays a screen showing game statistics such as player health, score, kills, or other relevant metrics.
- update_kill_count(self): Updates the number of enemies defeated by the player(s) and potentially triggers new objectives or level progression.
- draw_title(self): Draws the game's title screen, which may include the logo, start button, and other menu elements.
- `run(self)`: The game loop that constantly updates the screen, draws the players, and listens for events (like quitting the game).

- **Config**: This configuration code sets up the essential parameters for a 2D shooter game using **Pygame**. It defines the screen dimensions, player actions, game variables, and loads images and sounds that are used throughout the game.

  Key Attributes:

- `GRAVITY`: Controls the gravitational effect on characters and objects.
- `SCREEN_WIDTH`, `SCREEN_HEIGHT`: Defines the game window dimensions.
- screen_scroll, bg_scroll: This typically represents how much the screen (camera view) should move horizontally in a given frame and keeps track of the cumulative background scroll value.

- `ANIMATION_COOLDOWN`: Determines the delay between animation frames.
- `TILE_SIZE`: Defines the size of tiles for level design.
- level, MAXS_LEVELS: Defines the level and max level.
- `EXPLOSION_SPEED`: Sets the speed of explosion animations.
- `RED`, `GREEN`, `WHITE`, `BLACK`: Standardized RGB color values used throughout the game.
- `Bullet_group`, `grenade_group`, `explosion_group`, `enemy_group`, `item_box_group,decoration_group,water_group,exit_group`: Pygame sprite groups that handle game objects dynamically.

  Methods:

- `get_font()`: A static method that returns the default game font (`Futura`, size 30) for rendering text in the game UI.

- **Soldier**: The Soldier class is responsible for handling the movement, actions, animations, and interactions of a soldier character (could be a player or an AI enemy) in the game.

  Key Attributes :

- char_type: Specifies whether the character is a "player" or "enemy," which affects its behavior and interactions in the game.
- speed: Controls how quickly the soldier moves within the game world.
- ammo: Represents the number of bullets the soldier has available for shooting.
- `grenades`: Number of grenades the character can use.
- health: Defines the soldier's health points, determining how much damage they can take before dying.
- direction: Indicates the direction the soldier is facing (1 for right, -1 for left).
- jump: A boolean that indicates if the soldier is attempting to jump.
- `in_air`: Track jumping status.
- `animation_list`: Stores animation frames (e.g., Idle, Run, Jump, Death) for different actions and determines which animation should be shown.
- `bullet_group`: Manages bullets fired by the character.

- **World**: The World class is responsible for creating and managing the game environment or "level" based on the level data. It processes the data and spawns objects in the game world, like obstacles, items, enemies, the player, and other decorative elements.

  Supporting Classes:

- **Decoration**:
  - This class creates non-interactive decorative elements (e.g., bushes, small objects) in the world.
  - **Attributes**: It holds an image and a rect (rectangle for positioning) for each decoration object.
  - **Methods**:
    - __init__: Initializes the image and rect of the decoration.
    - update: Moves the decoration objects based on the scroll, so they follow the player's movement.
- **Water**:
  - This class represents water bodies in the world that can affect the player (e.g., drowning or health decrease).
  - **Attributes**: Like Decoration, it holds an image and rect for positioning.
  - **Methods**:
    - __init__: Initializes the water sprite with an image and position.
    - update: Moves the water sprites based on the screen scrolling.
- **Exit**:
  - This class represents the exit points of the level where the player can proceed to the next level.
  - **Attributes**: Similar to other classes, it holds the image and rect of the exit.
  - **Methods**:
    - __init__: Initializes the exit sprite with an image and position.
    - update: Moves the exit based on the screen scroll.

- **Item** : The Item class represents an item that the player can interact with in the game. These items include health boxes, ammo boxes. When the player collides with one of these item boxes, the player's stats (such as health, ammo) are updated accordingly.

  Methods:

- __init__: Initializes the item with a type, sets its image, and positions it properly within the game world.
- update: Checks for collision with the player. If the player touches the item, it applies the corresponding effect: **Health** - Increases the player's health (capped at max health). **Ammo** - Adds bullets to the player's ammo count. **Grenade** - Increases the player's grenade count. And Removes the item from the game after it is collected (`self.kill()`).

- **HealthBar** : The `HealthBar` class is used to visually represent a player's health on the screen. It creates and updates a health bar that is displayed on the game screen, showing how much health the player has remaining.

  Key Attributes:

- `x, y`: Position of the health bar on the screen.
- `health`: Current health of the player.
- `max_health`: Maximum possible health.

  Methods:

- `__init__:`Initializes the health bar with a position and tracks health values.
- `draw:`The draw method updates the health value, calculates the health ratio (health / max_health), and dynamically scales the green bar while drawing a black border (visibility), a red background (missing health), and a green bar (current health).

- **Bullet** : The `Bullet` class represents the bullets fired by the player. It handles the movement, collision detection, and removal of the bullet when it goes off-screen or hits an obstacle or enemy.

  Key Attributes:

- `speed`: Determines how fast the bullet moves per update cycle (default is `10`).
- `image`: Loads the bullet sprite (`img/icons/bullet.png`).
- `rect`: Defines the bullet's position and collision box.
- `direction`: Defines the bullet's movement direction (`1` for right, `-1` for left).

  Methods:

- `__init__:`Initializes the bullet at position (x, y), sets the bullet's direction (1 for right, -1 for left), loads the bullet image, and defines its rectangle (rect).
- update:Handles movement by moving the bullet in the specified direction (self.rect.x += self.direction * self.speed), off-screen removal by killing the bullet if it moves off the screen, and collision detection where if the bullet collides with the player, their health decreases by 5 points and the bullet is removed, or if it collides with an enemy, their health decreases by 25 points and the bullet is removed.

- **Level** : This `Level` class is a level editor for a 2D tile-based game built with Pygame. It allows users to create, edit, save, and load levels by placing tiles on a grid.

  Key Attributes:

- screen: The display surface for rendering the level.-
- world_data: A 2D list representing the tiles in the level.
- scroll: Controls horizontal scrolling.
- level: Tracks the current level number.
- img_list: Stores images of different tile types.
- save_button, load_button: Buttons for saving and loading level data.

  Methods:

- `draw_bg()`: Draws the background scenery.-
- `draw_grid()`: Displays a grid for level design.

- `draw_world()`: Renders the tiles of the level.
- `run()`: Main game loop, handling user interactions, scrolling, and tile placement.

- **Button**: The `Button` class is a simple button implementation using **Pygame**, allowing for interactive UI elements in a game or application. Here's a breakdown of how it works:

  Key Attributes:

- **self.image**: Stores the button's image, which is scaled based on the provided scale factor.

- **self.rect:** A pygame.Rect object representing the button's position and dimensions.

- **self.rect.topleft**: Sets the top-left position of the button on the screen.

- **self.clicked:** A boolean flag that prevents repeated activation when the button is held down.

Methods:

- `__init__:` This constructor scales the provided image, creates a rectangle to represent the button's position and size, sets its top-left corner at (x, y), and initializes self.clicked as False.
- draw:This method checks if the mouse is over the button and if it's clicked. It ensures only one click is registered per press, draws the button on the surface, and returns True if clicked, otherwise False.

- **StatisticsManager**: The `StatisticsManager` class is a utility class in Pygame used to track and manage gameplay statistics such as kills, deaths, grenades used, and more. It helps display these statistics on the game screen and allows other game objects to modify or reset values during runtime.

- Key Attributes:

- **self.stats**:
  A dictionary that holds key-value pairs representing different gameplay statistics (e.g., `'kills': 0`, `'grenades_used': 0`).
- **self.font**:
  A pygame.font.Font object used to render the statistic labels and values on the screen.
- **self.color**:
  The color used for drawing the text on the screen, typically in RGB format (e.g., white (255, 255, 255)).

Methods:

- __init__: The constructor initializes the statistics dictionary with default values (kills, deaths, etc.), sets the text color, and prepares the font used for rendering on-screen statistics.

- increment(): Increases the specified statistic by a given amount. For example, calling increment('kills') will add 1 to the kill count. A warning is printed if the stat key does not exist.

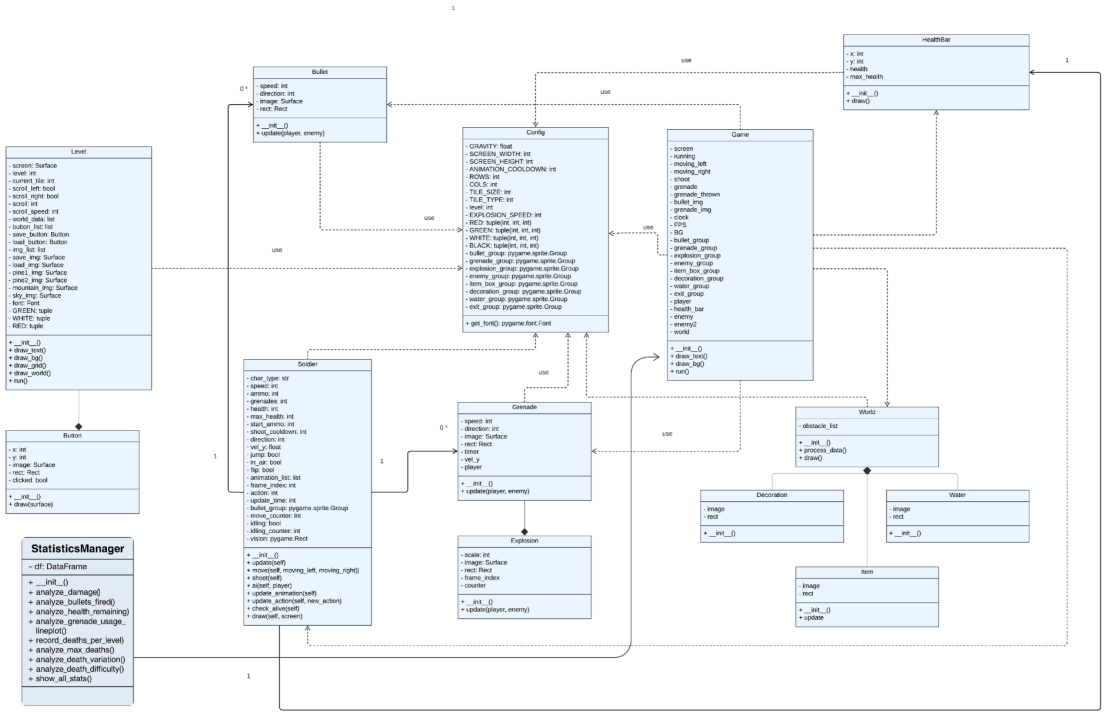
- set(Sets a specific statistic to a given value. This is useful for manual overrides or score adjustments.

- reset(): Resets all tracked statistics back to 0. Typically called at the beginning of a new game or level.

- get(stat_key):
  Returns the current value of a specific statistic. Returns None if the stat key is invalid.

- draw():Renders each statistic as a text label on the screen at the specified (x, y) location. Each line is spaced vertically by the line_spacing value.

UML Class Diagram: (*)



3.3 Algorithms Involved

- **Pathfinding Algorithm**: Enemies will use A* pathfinding to navigate terrain efficiently.
- **Collision Detection**: AABB (Axis-Aligned Bounding Box) method will detect interactions between player, enemies, and projectiles.
- **AI Behavior**: Rule-based decision-making will control enemy attacks and movement.
- **Event-driven mechanics**: Triggers for special events like power-up drops and boss fights.
- **Sorting Algorithm**: High scores will be sorted using quicksort or mergesort.

Statistical Data (Prop Stats)

4.1 Data Features

The game will track at least five key metrics, including:

1. **Damage Taken by the Player** – Helps analyze which enemies or obstacles cause the most damage, allowing developers to balance enemy attack power or defensive mechanics.

2. **Number of Bullets Fired** – Helps analyze shooting patterns and ammo consumption.
3. **Deaths per Level –** How many times the player died before completing a level.
4. **Health Remaining** – Captures player's final health at the end of a session.
5. **Number of Grenades Thrown** – Analyzes grenade usage frequency to balance grenade supply and explosion damage.

	Why is it good to have this data? What can it be used for?	How will you obtain 50 values of this feature data?	Which variable (and which class will you collect this from)?	How will you display this feature data (via summarization statistics or via graph)?
Damage Taken by the Player	Helps analyze which enemies or obstacles cause the most damage. Can balance enemy attack power or defensive mechanic	Track changes in self.player.health whenever it decreases.	self.player.`health` (from `Soldier` class).	Using a histogram can show the distribution of damage taken by the player over multiple sessions or levels.

Number of Bullets Fired	Helps analyze shooting patterns and ammo consumption. Can optimize ammo drop rates or adjust enemy difficulty.	Count how many times `self.player.shoot()` is called per minute and collect 50 values. (*)	`self.player.ammo` (from `Soldier` class).	Using scatter plot helps to identify if there's a correlation between the number of enemies and the number of bullets used, or how ammo consumption correlates with time spent in the game.
Deaths per Level	Helps measure game difficulty by identifying which levels cause the most player deaths and provides insights into whether levels are too easy, too hard, or just right.	Track self.player.deaths, record deaths per level across playthroughs, and gather data from multiple players or sessions for diverse results.	Variable: self.player.deaths Class: Soldier (or Player class)	Maximum Deaths: Identifies the hardest level. Standard Deviation: Measures variation in difficulty.
Health Remaining	Helps analyze game difficulty by tracking health loss patterns.Can be used to adjust enemy damage, health packs, or defensive mechanics.	Collect self.player.health every 5 seconds during gameplay.	`self.player.health` (from `Soldier` class).	Using box plot graphs to help determine which levels have the widest range of health outcomes, showing inconsistent difficulty.
Number of Grenades Thrown	Analyzes grenade usage frequency. Helps balance grenade	Track `self.grenade_thrown` each time a grenade is	`self.player.grenades` (from `Soldier` class).	A Line plot where grenades are used most often, helping identify levels

	supply and explosion damage.	thrown per minute. (*)		or enemies where grenades are frequently used.

4.2 Data Recording Method

Statistical data will be stored in a **CSV file** for easy retrieval and analysis.

Parameters:

- **1 Player** (Consistent skill level)
- **3 Levels** (Track deaths per level)
- **10-15 Playthroughs** (To get enough data points)

Record Deaths Per Level Per Playthrough:

- Each time the player dies on a level

Analyzing Difficulty with Maximum Deaths & Standard Deviation:

Maximum Deaths (Identifies the Hardest Level):

- Find the **level with the highest single death count** across all playthroughs.

Standard Deviation (Measures Variation in Difficulty):

- Standard deviation tells how spread out the deaths are for each level.
- A high standard deviation means some playthroughs had very high deaths while others had very low deaths, showing inconsistency.

If a level has high deaths + high standard deviation → It's **too difficult or inconsistent** and may need balancing.

if a level has low deaths + low standard deviation → It might be **too easy**.

Example Table: **Deaths Per Level** (Across Multiple Playthroughs)

Playthrough	Level 1 (Deaths)	Level 2 (Deaths)	Level 3 (Deaths)
1			
2			
3			
Total Deaths	x	y	z

4.3 Data Analysis Report

The recorded data will be analyzed using:

- **Mean, Median, Mode** for score distribution.
- **Progression Graphs** showing performance trends over multiple sessions.
- **Statistical tables** to observe trends in the value data over time.

	Feature Name	Graph Objective	Graph Type	X-axis	Y-axis
Graph 1	Damage Taken by the Player	Analyze **how much damage the player takes during gameplay** to identify trends, balance difficulty, and optimize enemy attack mechanics or defensive abilities.	Histogram	Damage value ranges (e.g., 0–25, 25–50, 50–75, etc.).	Frequency (number of times a specific range of damage is taken).
Graph 2	Number of Bullets Fired	Analyze **the player's shooting patterns and ammo consumption** to improve game balance and mechanics.	Scatter plot	Number of enemies faced or time (minutes played).	Number of bullets fired. (Frequency)
Graph 3	Number of Grenades Thrown	Analyze **how often players use grenades**, whether they rely on them too	Line plot	Time Intervals (e.g., 0-5 etc.)	Player categories (e.g., High Kills,

		much or too little, and how this affects overall gameplay.			Average Kills, Low Kills)
Graph 4	Health Remaining	Analyze how the remaining health of players correlates with their performance, survival time, or success in the game.	Box plot	Time Intervals (e.g., 0-5 etc.)	Health Remaining (percentage or hit points)

5. Project Timeline

Week	Task
1 (10 March)	Proposal submission / Project initiation
2 (17 March)	Full proposal submission
3 (24 March)	Initial game mechanics implementation
4 (31 March)	AI behavior & enemy design
5 (7 April)	Data collection & analysis system setup
6 (14 April)	Submission week (Draft)

5.1) Weekly planning: what I plan to do each week.

- **26 March-2 April :** Initial game mechanics implementation and AI behavior & enemy design (Done)
- **3 April-9 April :** Add some new features (e.g., sound effect , new items, new level etc.) (Done)
- **10 April-16 April :** Collect 50 data Features in csv file & analysis (Done)
- **17 April-23 April :** Examine code, identify any errors, and debug them. (Done)
- **24 April-11 May :** Submission all work.

5.2) List 50% of the tasks that are expected to be completed by 16 April.

- **Game Features**: Adding new features like sound effects, new items, new levels.
- **Data Collection**: Collect the 50 data points for deaths per level and complete the CSV file analysis.
- **Progress Report**: Having a clear report on the data collected and preliminary results for the first stage of analysis.

5.3) List 75% of the tasks that are expected to be completed by 23 April.

- **Data Analysis**: Completing in-depth analysis on the 50 data points you collected.
- **Debugging**: Start examining the code, identifying any issues, and addressing bugs or errors related to AI, enemy design, or game mechanics.
- **Feature Refinements**: Completing any missing features (if any) or refining the ones added earlier (e.g., improving AI behavior or balancing enemy design).
- **Testing**: Begin testing all implemented features and mechanics thoroughly to ensure functionality.

5.4) List the remaining 25% of the tasks that are expected to be completed by 11 May.

- **Code Review and Final Debugging:** Conduct a final review of the codebase to ensure that all features are working properly and no bugs are left.
- **Optimization:** Any optimizations needed in terms of performance, especially for the AI and game mechanics.
- **Final Testing:** Thorough testing of the entire game, ensuring all features, data collection, and AI work as expected.

- **Final Submission:** Submit the completed work, including documentation and any final reports on data analysis or other significant results.

6. Document version

Version: 5.0
Date: 10 May 2025

Kantapon Hemmadhun 6510545276

Note: (*) mean revision

Date	Name	Description of Revision, Feedback, Comments
15/3	Rattapoom	Very good! But don't forget to remove italics in your documents.
16/3	Parima	The proposal is clear with great detail. Good Work.
29/3	Parima	Game Concept misses some details and the diagram needs some revision.
29/3	Rattapoom	How are you going to get 50 rows of "Number of bullets fired" and "Number of Grenades Thrown"? If it's by playing the game 50 times then it'd take a long time depending on the length of your game. "Bullets fired per minute" may be more feasible.