HIMALAYAN COLLEAGE OF ENGINEERING

A Project Report

On

Survival Shooter RPG using Python

Submitted By:

Prerit Ghimire (HCE081BCT026)

Submitted To:

Department of Electronics and Computer Engineering

Himalaya College of Engineering

Chyasal,Lalitpur

Feb-23,2026

ACKNOWLEDGEMENT

We express our sincere gratitude to all those who have supported and guided us throughout the process of conducting this report on Survival Shooter RPG using Python. This endeavor would not have been possible without their valuable contributions and assistance.

The authors extend their heartfelt thanks to our supervisor, [Sushant Pandey], whose expertise and guidance have been instrumental in shaping the direction of this project. Their insightful feedback and continued encouragement have been invaluable in refining the scope and focus of this study.

We also would like to thank our friends and colleagues for their meaningful discussions, brainstorming sessions, and moral support. Your input has enriched the quality of this report and helped us navigate challenging concepts.

This work would not have been possible without the collective efforts of these individuals and organizations. Although any shortcomings in this report are solely our responsibility, their contributions have significantly enriched its content.

Thank you.

[Prerit Ghimire] ([HCE081BCT026])

ABSTRACT

This project focuses on developing a 2D Survival Shooter RPG using Python and Pygame. The game features a player-controlled character, auto-firing mechanics, homing enemy waves, item drop systems, experience and leveling mechanics, and an interactive menu system with mouse navigation. The primary goal of the project is to create a functional, engaging, and interactive game that demonstrates core concepts of game development including real-time rendering, sprite-based collision detection, wave-based difficulty scaling, and object-oriented design using Python. The game includes a complete UI flow: a main menu, a how-to-play screen, in-game HUD, and a game-over screen with persistent high-score tracking.

Table of context

List of Figures

List of Tables

List of Abbreviations

RPG — Role Playing Game
FPS — Frames Per Second
HUD — Heads Up Display
XP — Experience Points
HP — Hit Points
UI — User Interface
OOP — Object Oriented Programming
RGB — Red Green Blue

1. INTRODUCTION

Computer Graphics (CG) and game development focus on creating real-time interactive visual experiences using computers. Modern game engines and libraries rely on mathematical models of geometry, movement, and collision to produce a playable simulation rendered frame by frame. This report presents a mini project titled Survival Shooter RPG, which demonstrates core game development concepts such as sprite-based rendering, real-time input handling, vector-based projectile motion, wave-based enemy spawning, and an object-oriented design pattern using Python and Pygame.

1.1 Background Introduction

At a high level, a game loop converts input and state data into a rendered frame on screen, repeating 60 times per second. Even in 2D games, the same foundations appear: coordinate spaces, sprite collision detection, velocity vectors, and frame-rate-independent timing. Standard patterns such as the sprite group pattern, cooldown timer pattern, and the painter's algorithm for rendering are widely used to build efficient, playable games.

1.2 Motivation

Many beginner game development projects use high-level engines without understanding why rendering and collision work. This project is motivated by the need to build intuition from the ground up: how a sprite is drawn, how a bullet finds its direction, how enemy AI homes in on the player, and how a complete game loop is structured. Building without a heavy engine makes every mechanic explicit and educational.

1.3 Objectives

The main objectives of the project are listed below:

- Implement a fully functional 2D survival shooter game using Python and Pygame.
- Demonstrate real-time input handling, sprite-based rendering, and collision detection.
- Design a wave-based progression system with scaling enemy difficulty.
- Build a complete UI flow: main menu, how-to-play screen, in-game HUD, and game-over screen with persistent records.
- Apply object-oriented programming to organize game entities as sprite classes.

1.4 Scope

This project covers fundamental game development concepts used in 2D game programming:

- Sprite creation, group management, and automatic drawing via Pygame sprite groups.
- Real-time keyboard and mouse input handling using both event queue and get_pressed().
- Vector math for projectile direction using math.atan2, math.cos, and math.sin.
- Collision detection using pygame.sprite.spritecollide with kill-on-contact flags.
- XP, leveling, fire rate, and health progression systems.

It does not include sound, 3D rendering, advanced AI pathfinding, or network multiplayer. Those are suggested as future enhancement

2. LITERATURE REVIEW

2.1 Real-Time Game Loops

A game loop is the central pattern of all real-time games. Each iteration reads input, updates state, checks collisions, and renders the frame. Pygame's clock.tick(60) enforces a 60 FPS cap, making movement speeds hardware-independent. This pattern is documented widely in game development literature as the foundation of any interactive simulation.

2.2 Sprite-Based Rendering

Sprite-based rendering is the standard technique for 2D games. Each game object carries an image (a pixel buffer) and a rect (a position rectangle). Pygame's Group.draw() iterates all sprites and blits each image at its rect position onto the screen surface. This is analogous to the painter's algorithm: drawing background first, then foreground objects, with the screen cleared each frame to prevent smear artifacts.

2.3 Collision Detection

Axis-Aligned Bounding Box (AABB) collision is the simplest and most efficient form of collision detection for 2D sprite games. Pygame implements this via spritecollide(), which tests whether two sprites' rects overlap. While not pixel-perfect, AABB is sufficient for most 2D game mechanics and runs in O(n) time for a single sprite against a group.

2.4 Related Tools and Frameworks

The implementation is built using Python 3 and Pygame, a cross-platform 2D game library that wraps SDL (Simple DirectMedia Layer). Pygame handles display initialization, surface management, event polling, sprite abstraction, and timing. It is widely used in educational game development settings for its simplicity and directness. Alternative frameworks include Arcade (Python), SFML (C++), and MonoGame (C#).

3. METHODOLOGY

3.1 System Overview

The project is organized into five main modules:

- Sprite Module: Defines Player, Enemy, Bullet, and Item classes, each inheriting from pygame.sprite.Sprite.
- Group Module: Manages sprite groups (player group, enemy group, bullet group, item group) for batch update, draw, and collision testing.
- UI Module: Implements txt() and btn() helper functions for rendering centered text and interactive mouse-navigable buttons every frame.
- Screen Module: Contains main_menu(), how_to_play(), and game_over_screen() — each a self-contained event loop managing one screen state.
- Game Loop Module: game_loop() handles all real-time simulation: input, AI, physics, collision, item pickup, and rendering at 60 FPS.

3.2 Mathematical Foundation

3.2.1 Bullet Direction Vector

To fire a bullet from position (px, py) toward mouse target (mx, my), the angle is computed using:

```
angle = math.atan2(my - py, mx - px)
dx = math.cos(angle) * speed
dy = math.sin(angle) * speed
```

atan2 returns the angle in radians of the vector from origin to target, handling all four quadrants correctly. cos and sin decompose that angle into X and Y velocity components. The bullet then moves dx pixels horizontally and dy pixels vertically each frame, traveling in a straight line toward the target at constant speed regardless of direction.

3.2.2 Wave Scaling Formulas

Enemy attributes scale with wave number using linear formulas:

```
enemy_hp    = 30 + (wave - 1) * 5
enemy_speed = random.randint(1, 2) + wave // 5
enemies_per_wave = 5 + (wave - 1) * 2
```

HP increases by 5 per wave. Speed increases by 1 every 5 waves via integer division. Enemy count increases by 2 per wave, creating progressively denser and tougher waves

3.3 Algorithms Used

3.3.1 Cooldown timer

The fire rate limiter uses a timestamp comparison pattern: the time of the last shot is stored and compared against the current time each frame. If the elapsed time exceeds fire_rate (in milliseconds), a bullet is fired and the timestamp is updated. This avoids frame-rate-dependent counting and works correctly at any FPS.

3.3.2 XP and Level Progression

The XP threshold scales quadratically with level: a player at level L needs L * 50 XP to reach the next level. On reaching the threshold, XP resets to 0, level increments, the player gains 20 HP, and fire_rate decreases by 20ms (clamped at 80ms minimum). This creates a satisfying progression where each level is slightly harder to reach but rewards meaningfully.

3.3.3 Enemy Homing AI

Enemy movement uses axis-aligned homing: each frame, the enemy compares its x,y position to the player's and steps toward it by speed units on each axis independently. While simpler than true vector steering, this produces natural swarm clustering behavior as groups of enemies converge on the player from all sides.

3.4 Implementation Snippet

The following snippet shows the bullet direction calculation — the core mathematical operation of the game:

```
class Bullet(pygame.sprite.Sprite):

  def __init__(self, pos, target):

    super().__init__()

    self.image = pygame.Surface((8, 8))

    self.image.fill(YELLOW)

    self.rect = self.image.get_rect(center=pos)

    a = math.atan2(target[1]-pos[1], target[0]-pos[0])

    self.dx = math.cos(a) * 12

    self.dy = math.sin(a) * 12

    self.damage = 10
```

3.5 Screenshots / Figures

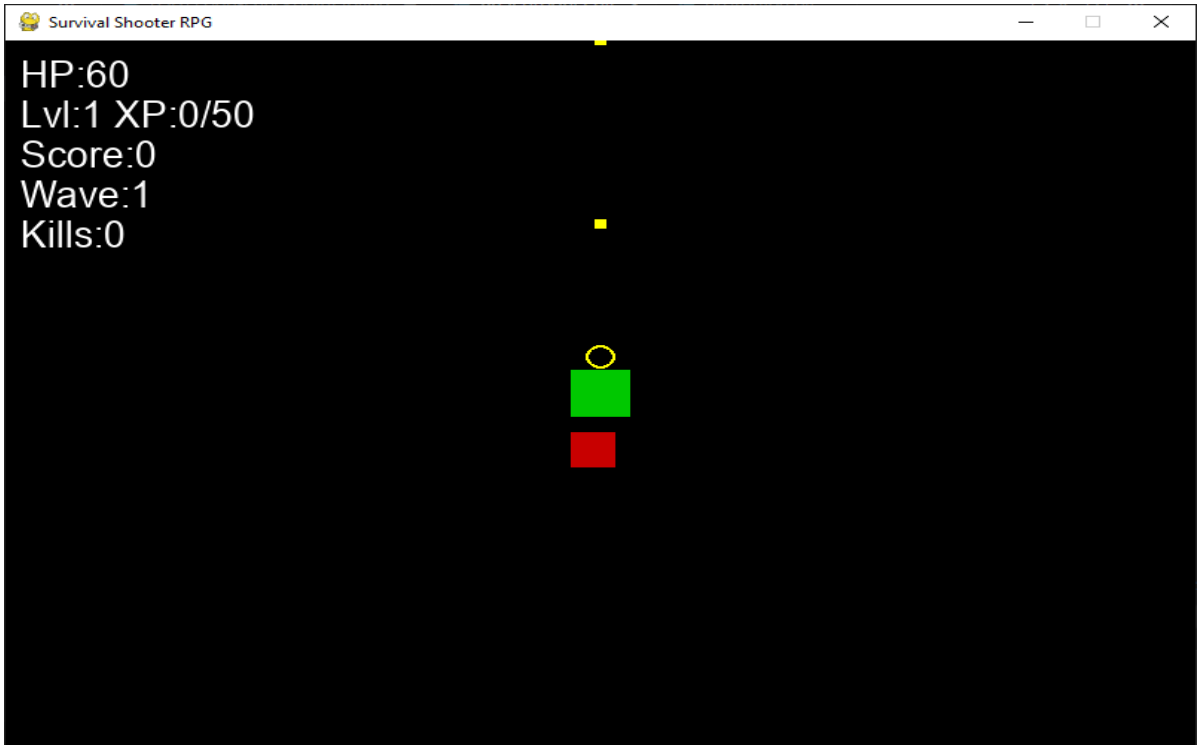

Figure 3.1: In-game HUD showing health bar, level, XP, score, wave, and kills

Figure 3.2: Game over screen displaying session stats and all-time high score records

3.6 Sample Table

Test case results for core game mechanics:

Test Case	Details	Observed Result
Main Menu	Mouse hover + click	Buttons highlight, navigate correctly
Wave Spawning	Waves 1-10	Enemy count, HP scale as expected
Bullet Collision	Single/multi enemy	Damage applied, enemy removed on 0 HP
Level Up	XP threshold at each level	Fire rate increases, HP bonus granted
Item Drops	30% drop chance	Health/XP pickups appear and apply effect
Game Over / Restart	Play Again + Menu button	Records saved, new game resets state

Table 3.1: Test case observations for game mechanIcs

4. RESULT AND ANALYSIS

4.1 Correctness of Game mechanics

All core game mechanics were verified through manual playtesting. Bullets consistently traveled in a straight line toward the mouse cursor position at the moment of firing. Enemy HP deducted correctly per bullet hit (10 damage per bullet), and enemies were removed from the group precisely at 0 HP. Item drops appeared at the correct position (enemy death coordinates) and applied their effects on player contact.

4.2 Gameplay Behavior

Interactive testing demonstrated correct behavior across all systems:

- Wave Progression: Enemies became demonstrably faster and required more hits to kill as wave numbers increased, consistent with the scaling formulas.
- Level Up: XP accumulated correctly across kills and item pickups. Level-up triggered at the expected threshold, with fire rate and HP updating immediately.
- UI Navigation: All menu buttons responded to both mouse click and hover correctly. The game-over screen displayed accurate session stats and correctly identified new personal bests.
- Record Persistence: High score, max level, and wave records persisted correctly across multiple runs within the same session.

4.3 Performance Discussion

The game maintained a stable 60 FPS throughout all tested scenarios. Pygame's sprite group system efficiently batched draw and update calls. The main performance bottleneck observed was at high wave numbers when 30+ enemies and 20+ bullets existed simultaneously, causing minor frame time spikes. The cooldown timer and boundary checks added negligible overhead. Text rendering (font.render()) was the costliest per-frame call due to surface creation, but remained within acceptable limits.

4.4 Limitations

- No anti-aliasing on sprites — all game objects are solid colored rectangles with sharp edges.
- No sound effects or background music — the game is entirely silent.
- Enemy AI uses simple axis-aligned homing with no pathfinding — enemies clip through each other.
- Records are session-only — high scores reset when the program is closed (no file persistence).
- No difficulty modes — the game has one fixed difficulty curve.

5. CONCLUSION AND FUTURE ENHANCEMENT

5.1 Conclusion

This project demonstrates the foundation of 2D game development by implementing a complete survival shooter game using Python and Pygame. Core game development concepts — the game loop, sprite-based rendering, AABB collision detection, vector-based projectile motion, and wave-based difficulty scaling — were successfully implemented and verified. The minimalist codebase (~110 lines without comments) proves that a fully functional game with menus, progression systems, and persistent records can be built with a clear understanding of fundamentals rather than reliance on complex engines.

5.2 Future Enhancements

- Sound: Add pygame.mixer for sound effects (gunshot, explosion, level-up) and background music.
- Persistent Records: Save high scores to a file using Python's json or pickle module so records survive between sessions.
- Sprite Art: Replace colored rectangles with actual sprite images loaded via pygame.image.load().
- Enemy Variety: Add different enemy types (fast-low-HP, slow-high-HP, ranged) with unique behaviors.
- Power-Ups: Add timed power-ups such as shield, rapid fire, or area explosion drops.
- Vector AI: Replace axis-aligned homing with true vector steering for more natural enemy movement.
- Anti-Aliasing: Implement smoother rendering using pygame.transform.smoothscale and alpha surfaces.

A. APPENDICES

A.1 Key Controls (Example)

- W / A / S / D: Move player up / left / down / right
- Mouse Movement: Aim the crosshair
- Auto-fire: Bullets fire automatically toward the mouse cursor
- ESC: Return to main menu from in-game
- Enter: Confirm selection in menu screens
- Mouse Click: Navigate all menu buttons

A.2 Additional Screenshot

Figure A.1: Main menu with mouse-navigable PLAY, HOW TO PLAY, and QUIT buttons

Bibliography

[1] Pygame Documentation, Pygame Community, https://www.pygame.org/docs/, Accessed 2025.

[2] A. Sweigart, Making Games with Python & Pygame, Creative Commons, 2012.

[3] R. Nystrom, Game Programming Patterns, Genever Benning, 2014.

[4] Python Software Foundation, Python 3 Documentation, https://docs.python.org/3/, Accessed 2025.