



**TRIBHUVAN UNIVERSITY  
INSTITUTE OF ENGINEERING**

**A Project Report**

**On**

**RUNFORGE:**

**Design and Development of a Modular Endless Runner Game  
Using Python and Pygame**

**Submitted By:**

Krrish Nyupane (HCE081BEI019)

**Submitted To:**

Department of Electronics and Computer Engineering

Himalaya College of Engineering

Chyasal,Lalitpur

Feb-22-2026

## **ACKNOWLEDGEMENT**

I express my sincere gratitude to all those who supported and guided me throughout the process of conducting this report on RUNFORGE: Design and Development of a Modular Endless Runner Game Using Python and Pygame. This project would not have been possible without their valuable guidance, encouragement, and contributions.

I extend my heartfelt thanks to my supervisor, Sushant Pandey, whose expertise and constructive feedback played a crucial role in shaping the technical direction and overall structure of this project. His insights into software design, problem-solving approaches, and system implementation greatly enhanced the quality and depth of this work.

I would also like to express my appreciation to my friends and colleagues for their meaningful discussions, brainstorming sessions, and continuous moral support throughout the development process. Their feedback helped refine gameplay mechanics, improve system performance, and strengthen the final implementation.

Finally, I acknowledge the open-source community and the developers of Python and Pygame, whose tools and documentation made the development of RUNFORGE possible.

Although any shortcomings in this report remain solely my responsibility, the guidance and support I received significantly enriched both the project and this documentation.

Thank you.

Krrish Nyupane (HCE081BEI019)

## ABSTRACT

It is a modular 2D endless runner game developed using Python and the Pygame library, designed to demonstrate practical game development concepts, real-time system design, and object-oriented programming principles. The project focuses on creating a scalable and reusable architecture that supports smooth gameplay, dynamic difficulty progression, and interactive visual and audio feedback. The game features automatic player movement, obstacle avoidance, falling hazards, a health-based survival system, power-up mechanics, and persistent score tracking to enhance player engagement.

The development process emphasizes modular design, where core components such as player control, obstacle generation, collision handling, rendering, and game state management are implemented as independent yet interconnected modules. This approach improves code readability, maintainability, and future extensibility. Visual effects, including animations, particle systems, parallax backgrounds, and modern UI elements, are integrated to provide an immersive gameplay experience, while audio feedback enhances responsiveness and player interaction.

RUNFORGE also incorporates performance optimization techniques to maintain a stable frame rate and ensure smooth execution across different systems. The project demonstrates how Python and Pygame can be effectively used to develop real-time interactive applications beyond basic prototypes. Overall, this work serves as a comprehensive case study in designing and implementing a modular endless runner game while balancing performance, usability, and software engineering best practices.

## Table of Contents

<b>ACKNOWLEDGMENT . . . . .</b>	<b>i</b>
<b>ABSTRACT . . . . .</b>	<b>ii</b>
<b>1. Introduction . . . . .</b>	<b>1</b>
1.1 Background Introduction . . . . .	1
1.2 Motivation . . . . .	1
1.3 Objectives . . . . .	1
1.4 Scope . . . . .	2
<b>2. Literature Review . . . . .</b>	<b>3</b>
2.1 2D Rendering and Raster Graphics . . . . .	3
2.2 Animation and Frame-Based Rendering . . . . .	3
2.3 2D Transformations and Coordinate Systems . . . . .	3
2.4 Collision Detection in 2D Graphics . . . . .	3
2.5 Particle Systems and Visual Effects . . . . .	4
2.6 Parallax Scrolling and Depth Illusion . . . . .	4
2.7 Graphics Frameworks and Game Development Libraries . . . . .	4
<b>3. Methodology . . . . .</b>	<b>5</b>
3.1 System Overview . . . . .	5
3.2 Game Loop Design . . . . .	5
3.3 Mathematical Foundation . . . . .	5
3.3.1 Gravity and Jump Mechanics . . . . .	5
3.3.2 Rotation Transformation . . . . .	6
3.3.3 Parallax Scrolling . . . . .	6
3.4 Algorithms Used . . . . .	6
3.4.1 Frame-Based Sprite Animation . . . . .	6
3.4.2 Collision Detection (AABB) . . . . .	6
3.4.3 Particle System . . . . .	7
3.5 User Interface Design . . . . .	7
3.6 Performance Strategy . . . . .	7
3.7 Testing and Validation . . . . .	7
<b>4. Result and Analysis . . . . .</b>	<b>8</b>
4.1 Rendering and Visual Output . . . . .	8
4.2 Gameplay Mechanics Behavior . . . . .	8
4.3 User Interface and Interaction Analysis . . . . .	8
4.4 Performance Discussion . . . . .	8

4.5	Limitations	9
<b>5.</b>	<b>Conclusion and Future Enhancement</b>	<b>10</b>
5.1	Conclusion	10
5.2	Future Enhancements	10
<b>A.</b>	<b>Appendices</b>	<b>11</b>
A.1	Game Controls	11
A.2	Additional Screenshot	11
	<b>Bibliography</b>	<b>12</b>

## **1. INTRODUCTION**

JUMPING ENGINEER is a 2D endless side-scrolling runner game developed using Python and the Pygame library as part of a Computer Graphics project. The game demonstrates practical implementation of core computer graphics concepts such as real-time rendering, animation, coordinate transformation, collision detection, parametric curve drawing, gradient generation, particle systems, and user interface design. The player controls a running character who must jump over obstacles and avoid falling asteroids while the difficulty increases dynamically based on score. The project integrates graphical effects like parallax scrolling, glassmorphism UI, glow effects, sprite animation, and procedural background generation, making it both a technical and creative demonstration of applied computer graphics principles.

### **1.1 Background Introduction**

Computer Graphics focuses on generating and manipulating visual content using programming techniques. Modern 2D games rely heavily on rendering pipelines, animation frames, transformation matrices, physics simulation, and user interaction handling. Using the Pygame framework, this project applies foundational graphics concepts such as frame-based animation, double buffering, color gradients, alpha blending, trigonometric curve drawing (heart parametric curve), and real-time event handling. The project bridges theoretical graphics concepts with practical implementation in an interactive game environment.

### **1.2 Motivation**

The motivation behind developing JUMPING ENGINEER was to practically apply computer graphics concepts in a real-time interactive system rather than limiting learning to theory. Instead of creating static graphical demonstrations, the goal was to build a complete playable game that integrates animation, physics, UI design, and rendering optimization. This project also aims to understand how modern 2D games manage performance, handle assets, implement visual effects, and maintain smooth 60 FPS rendering. Additionally, it was motivated by the challenge of designing a visually appealing and technically structured game entirely in Python.

### **1.3 Objectives**

The main objectives of the project are listed below:

- To implement real-time 2D rendering using Python and Pygame.
- To apply computer graphics concepts such as transformations, animation, gradients, and alpha blending.
- To design sprite-based character animations with multiple states.
- To implement collision detection between dynamic objects.
- To develop procedural background generation with parallax scrolling.
- To create particle effects (fire, smoke, sparks) using basic physics principles.
- To design an interactive graphical user interface with glow and glass effects.
- To implement a structured game state machine.
- To maintain a stable 60 FPS rendering performance.
- To implement persistent data storage using JSON files.

## 1.4 Scope

This project covers fundamental CG concepts used in 2D rendering:

- The project focuses exclusively on 2D graphics rendering.
- It supports single-player offline gameplay.
- It demonstrates dynamic difficulty scaling mechanisms.
- It includes multiple map environments with procedural background design.
- It integrates audio, particle systems, and UI effects within the rendering pipeline.
- The architecture allows future expansion such as additional levels, enemies, or leaderboard integration.
- The project can be extended to executable builds or mobile adaptations.

It does not include a full 3D pipeline, advanced shading (Phong/PBR), or GPU shader programming. Those are suggested as future enhancements.

## 2. LITERATURE REVIEW

### 2.1 2D Rendering and Raster Graphics

Raster graphics systems convert geometric and mathematical representations into pixel-based images displayed on the screen. In real-time 2D games, rendering involves drawing sprites, shapes, gradients, and text onto a frame buffer at a fixed refresh rate. Double buffering techniques are commonly used to prevent flickering and ensure smooth animation. In *JUMPING ENGINEER*, the rendering pipeline updates and redraws all graphical elements at 60 frames per second using Pygame's surface-based raster rendering system.

### 2.2 Animation and Frame-Based Rendering

Frame-based animation is a fundamental concept in computer graphics where motion is simulated by displaying a sequence of images in rapid succession. Sprite animation is widely used in 2D games to represent character states such as running or jumping. The illusion of motion is achieved through continuous frame updates synchronized with the game clock. The project applies sprite-sheet animation techniques for the player character and combines them with time-based state transitions.

### 2.3 2D Transformations and Coordinate Systems

2D transformations such as translation, scaling, and rotation are essential for positioning and animating graphical objects. These transformations are often represented using matrix operations in homogeneous coordinates, enabling multiple transformations to be combined efficiently. In this project, translation is used for obstacle movement, rotation is applied to falling asteroids, and scaling is used for countdown effects and UI animations. The Cartesian coordinate system of the display window is used as the reference frame for all object positioning.

### 2.4 Collision Detection in 2D Graphics

Collision detection determines when two graphical objects intersect within a coordinate space. In 2D systems, bounding box (Axis-Aligned Bounding Box) techniques are commonly used for efficient collision checks. This method compares rectangular boundaries of objects to detect overlaps. *JUMPING ENGINEER* uses rectangle-based collision detection between the player, obstacles, and asteroids to ensure accurate interaction and gameplay mechanics.

## **2.5 Particle Systems and Visual Effects**

Particle systems simulate fuzzy phenomena such as fire, smoke, and sparks by rendering many small graphical elements with randomized motion and lifetimes. These systems enhance visual realism and user feedback in interactive applications. The project implements particle effects using physics-inspired motion updates, gravity simulation, and alpha transparency blending to create dynamic visual feedback during jumps, shield activation, and collisions.

## **2.6 Parallax Scrolling and Depth Illusion**

Parallax scrolling is a graphical technique where background layers move at different speeds relative to the foreground, creating an illusion of depth in 2D environments. This technique is widely used in side-scrolling games to simulate three-dimensional perception within a two-dimensional space. In *JUMPING ENGINEER*, multiple background layers such as mountains, trees, clouds, and dunes move at fractional speeds of the player's movement to achieve depth perception.

## **2.7 Graphics Frameworks and Game Development Libraries**

Modern 2D applications and games are developed using graphics frameworks such as OpenGL, SDL, Unity, or Pygame. While hardware-accelerated APIs provide high-performance rendering, educational frameworks like Pygame offer simplicity and clarity for understanding graphics fundamentals. The project utilizes Pygame for surface rendering, event handling, sprite animation, and audio integration, enabling a structured and educational implementation of computer graphics principles.

### 3. METHODOLOGY

#### 3.1 System Overview

The system is organized into modular components to maintain clarity, scalability, and structured game flow:

- **Game Core Module:** Manages the main game loop, state transitions (start menu, playing, pause, game over), and overall control flow.
- **Rendering Module:** Handles real-time drawing of sprites, backgrounds, UI elements, gradients, and particle effects at 60 FPS.
- **Physics and Collision Module:** Implements gravity, jumping mechanics, obstacle movement, and rectangle-based collision detection.
- **UI and Interaction Module:** Manages keyboard/mouse inputs, animated buttons, glow effects, and in-game HUD elements.
- **Persistence Module:** Stores best scores and settings using JSON-based file handling.

#### 3.2 Game Loop Design

The application follows a structured real-time loop controlled by a fixed frame rate:

```
while running:  
    handle_events()  
    update()  
    draw()  
    clock.tick(60)
```

This loop ensures consistent frame updates, stable animations, and synchronized input processing.

#### 3.3 Mathematical Foundation

##### 3.3.1 Gravity and Jump Mechanics

The vertical motion of the player follows basic kinematic principles:

$$v = v + g$$

$$y = y + v$$

where  $v$  is vertical velocity and  $g$  is gravity acceleration applied each frame. This produces a smooth parabolic jump motion.

### 3.3.2 Rotation Transformation

Falling asteroids rotate using incremental angle updates:

$$\theta = \theta + \omega$$

where  $\omega$  represents angular velocity. The rotated sprite is rendered using surface transformation functions.

### 3.3.3 Parallax Scrolling

Background layers move at fractional speeds relative to player speed:

$$x_{layer} = x_{layer} - (speed \times factor)$$

Different factors (e.g., 0.1, 0.3, 0.6) create a depth illusion.

## 3.4 Algorithms Used

### 3.4.1 Frame-Based Sprite Animation

Character animation is achieved by cycling through preloaded sprite frames at timed intervals. The animation index is updated every few frames to simulate motion.

### 3.4.2 Collision Detection (AABB)

Axis-Aligned Bounding Box (AABB) collision detection is used:

Two rectangles collide if:

$$x_1 < x_2 + w_2 \quad \text{and} \quad x_1 + w_1 > x_2$$

$$y_1 < y_2 + h_2 \quad \text{and} \quad y_1 + h_1 > y_2$$

This method is computationally efficient and suitable for real-time 2D systems.

### **3.4.3 Particle System**

Each particle maintains position, velocity, size, and lifetime. At every frame:

- Position is updated using velocity.
- Gravity is applied if required.
- Size decreases gradually.
- Particle is removed when lifetime expires.

This creates fire, spark, and smoke effects during gameplay.

### **3.5 User Interface Design**

The graphical interface uses layered rendering techniques:

- Semi-transparent surfaces for glassmorphism effects.
- Gradient fills for backgrounds.
- Glow effects using alpha-blended overlays.
- Outlined text rendering for readability over dynamic backgrounds.

### **3.6 Performance Strategy**

To maintain stable performance:

- Rendering is capped at 60 FPS using `clock.tick(60)`.
- Double buffering prevents flickering.
- Object lists (obstacles, particles) are cleaned each frame.
- Asset loading occurs once during initialization.

### **3.7 Testing and Validation**

- Collision detection tested with boundary debugging.
- FPS monitored to ensure stable performance.
- Asset-missing fallback mechanisms verified.
- Score persistence tested across multiple sessions.

## 4. RESULT AND ANALYSIS

### 4.1 Rendering and Visual Output

The game successfully renders real-time 2D graphics at a fixed resolution of  $1100 \times 700$  pixels while maintaining a stable 60 FPS under normal gameplay conditions. Sprite animations appear smooth due to frame-based updates synchronized with the game clock. The parallax scrolling system creates a depth illusion by moving background layers at fractional speeds relative to the player. Gradient skies, glow effects, glassmorphism panels, and particle effects enhance visual quality while remaining computationally efficient. Minor aliasing is visible on diagonal edges due to raster-based rendering, which is expected in pixel displays without anti-aliasing.

### 4.2 Gameplay Mechanics Behavior

The physics-based jump system produces a smooth parabolic motion due to incremental gravity updates applied each frame. Collision detection using Axis-Aligned Bounding Boxes (AABB) accurately detects intersections between the player, obstacles, and asteroids. The dynamic difficulty scaling mechanism increases game speed proportionally to the score, providing progressive challenge without abrupt transitions. Shield activation and invincibility frames function correctly by temporarily disabling collision damage, demonstrating proper state management.

### 4.3 User Interface and Interaction Analysis

The interactive UI responds instantly to keyboard and mouse inputs. Glow effects and hover animations provide visual feedback for button interactions. Semi-transparent panels and outlined text improve readability over dynamic backgrounds. The structured state machine ensures smooth transitions between start menu, gameplay, pause screen, settings, and game over states.

### 4.4 Performance Discussion

Performance testing shows stable 60 FPS during normal gameplay. Under high particle generation scenarios, frame rate slightly fluctuates between 58–60 FPS, which remains visually smooth. Optimization strategies such as:

- Preloading assets during initialization,
- Removing off-screen objects,
- Using rectangle-based collision checks,

- Limiting particle lifetimes,

help maintain performance stability. Double buffering prevents screen flickering and ensures smooth frame updates.

#### 4.5 Limitations

- No hardware-accelerated rendering (software-based Pygame surfaces only).
- Basic rectangle-based collision detection (no pixel-perfect collision).
- No advanced lighting or shading models.
- Limited to 2D rendering without 3D perspective simulation.

## 5. CONCLUSION AND FUTURE ENHANCEMENT

### 5.1 Conclusion

The project *JUMPING ENGINEER* successfully demonstrates practical implementation of 2D computer graphics concepts within an interactive game environment. It integrates real-time rendering, sprite animation, transformation techniques, collision detection, parallax scrolling, and particle systems into a structured and modular architecture. The system maintains stable performance while delivering visually engaging effects, validating the application of theoretical computer graphics principles in real-world software development.

### 5.2 Future Enhancements

- **Anti-aliasing Implementation:** Apply smoothing techniques for improved visual quality.
- **Advanced Collision Detection:** Implement pixel-perfect or mask-based collision systems.
- **Lighting and Shadow Effects:** Introduce dynamic lighting models for improved realism.
- **Boss Enemies and AI Behavior:** Add intelligent enemy movement algorithms.
- **Online Leaderboard System:** Integrate network-based score tracking.
- **Mobile Porting:** Adapt controls and rendering for touchscreen devices.
- **3D Perspective Simulation:** Extend parallax and scaling techniques for pseudo-3D depth.

## A. APPENDICES

### A.1 Game Controls

- **SPACE / W / UP Arrow:** Jump / Double Jump
- **S:** Activate Shield
- **P or ESC:** Pause / Resume
- **R:** Restart (Game Over)
- **M:** Return to Main Menu
- **Mouse Click:** Interact with UI buttons

### A.2 Additional Screenshot

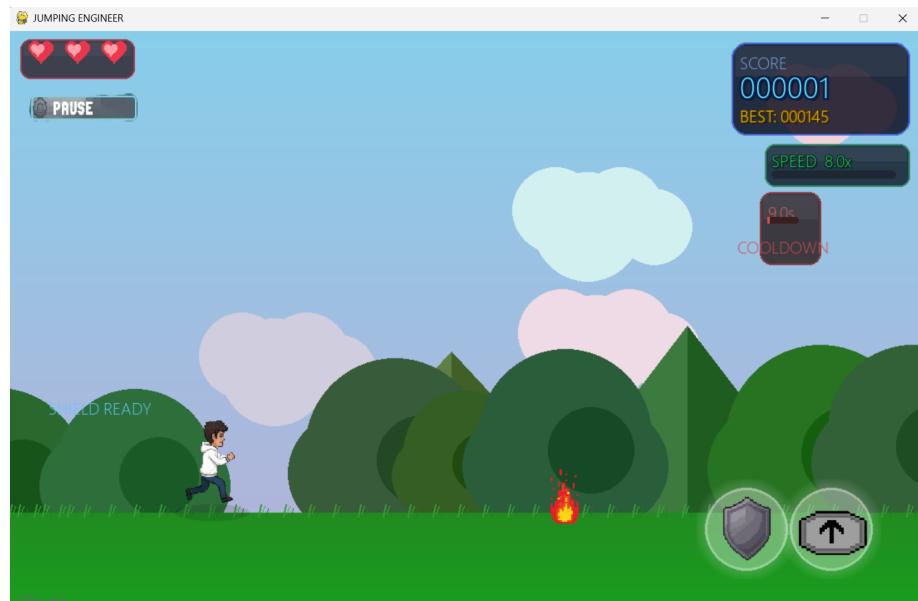


Figure A.1: Gameplay screen showing player, obstacles, parallax background, and UI

## Bibliography

- [1] sentdex, *Game Development in Python 3 With PyGame - Tutorial Series*, YouTube video, 2014. Available: <https://www.youtube.com/watch?v=ujOTNg17LjI> :contentReference[oaicite:1]index=1
- [2] Tech with Tim, *Python Game Development: From Beginner to Advanced*, YouTube playlist/course, 2025. Available via ClassCentral. :contentReference[oaicite:2]index=2
- [3] Keith Galli, *How to Program a Game in Python*, YouTube video, 2025. Available via ClassCentral. :contentReference[oaicite:3]index=3
- [4] Python Simplified, *Create a Simple Video Game with Pygame – Step-by-Step Tutorial*, YouTube video, 2025. Available via ClassCentral. :contentReference[oaicite:4]index=4
- [5] GeeksforGeeks, *PyGame Tutorial*, Online tutorial, 2025. Available: <https://www.geeksforgeeks.org/pygame-tutorial/> :contentReference[oaicite:5]index=5
- [6] Real Python, *Python Game Development Tutorials*, Online resource, 2026. Available: <https://realpython.com/tutorials/gamedev/> :contentReference[oaicite:6]index=6
- [7] Boyao Song, *Game Development with Python Using Pygame*, in Proceedings of the International Conference on Information Economy, Data Modeling and Cloud Computing (ICIDC), 2022. DOI: 10.4108/eai.17-6-2022.2322877. :contentReference[oaicite:7]index=7
- [8] GitHub Copilot, AI Code Assistant, GitHub Inc., 2023–2026. Used for code suggestions and development assistance.
- [9] Gemini AI, AI Image Generation Tool, Google DeepMind, 2025–2026. Used to generate model graphics and UI assets.
- [10] Pygame, in Wikipedia, The Free Encyclopedia, 2025. Available: <https://en.wikipedia.org/wiki/Pygame>