iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

WASTELAND RUN

A Post-Apocalyptic Endless Runner: Full-Stack Web Game with Flask Backend, Procedural Canvas Animation, and Leaderboard System

J. Allan Dericks & E. Karthi
UG Students, Department of Computer Applications, VISTAS, Pallavaram, Chennai, India
Dr. M. Sakthivanitha
Assistant Professor, Department of Computer Applications, VISTAS, Pallavaram, Chennai, India

Abstract
Wasteland Run is a browser-based post-apocalyptic endless runner developed as a final year project. The game is implemented using a Python Flask backend for score persistence and leaderboard management, and a vanilla JavaScript HTML5 Canvas frontend that renders all visual content procedurally without external assets. The system features a layered parallax background with animated ruins and a blood moon, a six-state procedural character animation engine, three interactive obstacle and collectible types, a particle effects system, and a combo-based scoring mechanic. Game state is managed through a real-time loop with delta-time physics, level progression, screen shake, and invincibility frames. The leaderboard stores the top ten scores in a JSON flat file via a REST API, accessible to all players. This paper documents the system architecture, rendering pipeline, physics implementation, API design, and validation results.

Keywords
Endless Runner; HTML5 Canvas; Flask; Procedural Animation; Particle System; Parallax Rendering; Leaderboard API; JavaScript Game Development; Post-Apocalyptic.

1. INTRODUCTION

Browser-based games offer an accessible medium for demonstrating full-stack development skills: the client must deliver responsive, visually rich real-time interaction while the server handles persistence and shared state. Endless runner games impose additional constraints, requiring smooth 60 fps animation, predictable physics, progressive difficulty, and reliable collision detection, all within the sandboxed browser environment.

Wasteland Run was designed to satisfy these constraints using only the browser's native Canvas 2D API for rendering and Python Flask for the backend. The decision to avoid sprite sheets, game engines, and pre-rendered assets was deliberate: every visual element is drawn procedurally each frame, demonstrating low-level graphics programming and algorithmic animation in addition to web development skills.

The principal contributions of this project are:

- A procedural character animation system with six distinct states (IDLE, RUN, JUMP, DOUBLE_JUMP, DUCK, HURT, LAND) rendered entirely through layered canvas primitives.
- A multi-layer parallax background engine generating scrolling post-apocalyptic ruins, a blood moon, stars, and a ground plane with animated cracks and dust.
- A physics engine with gravity, variable jump velocities, double-jump, invincibility frames, and delta-time normalisation for frame-rate independence.
- A particle system supporting four effect types: coin burst, hit sparks, dust puff, and level-up ring.
- A REST leaderboard API built with Flask that persists scores to JSON, returns the top-ten rankings, and serves the single-page application.

The remainder of this document is organized as follows. Section 2 reviews related work. Section 3 describes the system architecture. Section 4 details the technology stack. Section 5 explains the game mechanics and rendering pipeline. Section 6 documents the REST API. Section 7 presents testing and results. Section 8 covers applications. Section 9 concludes with future work.

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

2. RELATED WORK

2.1 Browser-Based Game Frameworks

Commercial JavaScript game engines such as Phaser, Babylon.js, and Three.js provide asset management, physics integration, and rendering abstractions. While effective for production titles, these frameworks obscure the rendering pipeline and physics mathematics. Wasteland Run instead uses the native Canvas 2D API directly, following the approach of educational projects that prioritise transparency over convenience.

2.2 Procedural Animation in Canvas Games

Most browser runner games rely on pre-drawn sprite sheets loaded as image assets. Procedural canvas drawing for character animation, as employed here, is less commonly documented and requires explicit management of joint positions, limb angles, and animation state transitions using trigonometric functions and easing curves.

2.3 Score Persistence in Lightweight Web Games

Persistent leaderboards in browser games commonly use localStorage for single-player high-score tracking, or cloud databases for shared rankings. The MSH project demonstrates that a Flask-backed JSON file provides a pragmatic middle ground: shared persistence without database overhead, suitable for small deployments such as academic final year projects or classroom demonstrations.

3. SYSTEM ARCHITECTURE

Wasteland Run follows a client-server model with a clear separation between the rendering and game logic layer in the browser and the score persistence layer on the server.

3.1 File Structure

File	Layer	Responsibility
app.py	Backend	Flask server — score submission, leaderboard, static file serving
index.html	View	Jinja2 template — HUD, canvas elements, overlays, script imports
game.js	Game Logic	Main loop, physics, collision, scoring, level progression
character.js	Rendering	Six-state procedural character animation engine
entities.js	Rendering	Burnt car obstacle, pterodactyl enemy, spinning coin
particles.js	Rendering	Particle system: burst, spark, square, text, dust effects
background.js	Rendering	Three-layer parallax background with moon, stars, ruins, ground
style.css	UI	HUD layout, overlay panels, leaderboard, responsive mobile styles
scores.json	Persistence	Flat-file leaderboard storage (auto-created on first run)

3.2 Canvas Layer Architecture

The game renders across three stacked canvases, each at 750 × 280 pixels. This separation allows each layer to be cleared and redrawn independently, avoiding unnecessary full-scene redraws.

- bgCanvas (z-index 1): Background layer drawn by Background class. Cleared and redrawn each tick at parallax-adjusted offsets.
- gameCanvas (z-index 2): Obstacles, collectibles, and the player character drawn by game.js each frame.
- fxCanvas (z-index 3): Particle effects drawn by ParticleSystem. Pointer-events disabled so clicks pass through to game canvas.

3.3 Backend Architecture

The Flask backend exposes three routes. The root GET / serves the Jinja2 template. POST /score accepts a JSON body containing a player name and score, updates the global high score, appends the entry to the leaderboard

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

array, sorts and trims it to ten entries, persists the updated structure to scores.json, and returns the high score and full leaderboard. GET /leaderboard returns the current scores without modification.

4. TECHNOLOGY STACK

Layer	Technology	Purpose
Backend	Python 3 / Flask ≥3.0	REST API, score persistence, static file serving
Persistence	JSON flat file	Leaderboard storage without database dependency
Frontend View	Jinja2 / HTML5	Single-page template, canvas elements, HUD
Rendering	Canvas 2D API	All game graphics drawn procedurally each frame
Game Logic	Vanilla JavaScript	Physics, collision, state machine, scoring, level progression
Styling	CSS3	HUD layout, overlays, scanline effect, responsive breakpoints
Typography	Orbitron / Share Tech Mono	Post-apocalyptic HUD aesthetic via Google Fonts
API Client	fetch (native)	Score submission and leaderboard retrieval

5. GAME MECHANICS AND RENDERING PIPELINE

5.1 Main Game Loop and Physics

The game loop is driven by requestAnimationFrame. Each tick computes a delta time (capped at 50 ms to prevent large jumps after tab suspension) and updates all subsystems proportionally, ensuring consistent physics at any frame rate.

Player physics uses a simple vertical velocity model. On jump, playerVY is set to JUMP_VEL (−13 px/frame at 60 fps). Each tick gravity (0.55) is added to playerVY and playerY is updated accordingly. A second jump applies JUMP2_VEL (−11). On landing, both values are reset, the jumps counter clears, and a LAND state triggers a squash animation and dust particle burst.

5.2 Character Animation State Machine

The Character class manages a state machine with six values defined in the CharState enum. Each state drives a set of pose parameters: body vertical bob, lean angle, leg swing phase, arm swing phase, head bob amplitude, crouch factor, air spin rotation, and jump stretch. These parameters feed directly into canvas transforms applied to each body segment during draw, producing smooth procedural animation without keyframes or sprite data.

State	Trigger	Visual Behaviour
IDLE	Before game starts	Slow breathing bob, gentle arm sway
RUN	Game active, on ground	Leg and arm swing cycle, forward lean, dust exhaust
JUMP	First space press	Body stretch, arms raised, forward lean increases
DOUBLE_JUMP	Second space press in air	Full body spin rotation, sparkle particles
DUCK	Arrow down / S held	Crouch with scaleY reduction, reduced hitbox height
HURT	Collision without invincibility	Lean oscillation, white flash overlay, red eye render
LAND	Touching ground after jump	Squash/stretch via scaleX/scaleY, auto-exits to RUN

5.3 Entities and Collision

Three entity types are managed in the game state object. The burnt car spawns at ground level and is drawn procedurally with animated flame tongues, rising smoke puffs, and a shadow ellipse. The pterodactyl spawns at

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

one of two heights (100 or 170 px above ground) in Zone 2 and above, moves faster than the car, and oscillates vertically. The coin spawns at a random height between 80 and 140 px above ground.

Collision detection uses axis-aligned bounding box (AABB) intersection. The player hitbox shrinks from PLAYER_H (62 px) to DUCK_H (36 px) when ducking. Coin collection adds a combo-scaled bonus and triggers a coin burst particle effect. Enemy collisions decrement lives and trigger 900 ms of invincibility with a 80 ms flicker effect.

5.4 Parallax Background

The Background class pre-generates building silhouette arrays at construction time with randomised widths, heights, window positions, antenna flags, and rubble flags. At runtime, each building layer is drawn offset by a parallax factor (0.25 for far ruins, 0.55 for mid ruins) applied to the global scroll offset. The moon, stars with per-star twinkle phase, toxic horizon haze, and animated ground cracks complete the scene.

5.5 Level Progression and Scoring

Level is derived directly from score: level = min(10, 1 + floor(score / 600)). Each level increases obstacle speed by 0.38 px/frame and background scroll speed by 0.30 px/frame. Score increments each tick by round(level × 0.12 + 0.5), giving a modest acceleration of score gain at higher levels. Coin bonuses award 50 + combo × 10 points, rewarding consecutive collections.

6. REST API ENDPOINT CATALOGUE

Method	Endpoint	Access	Description
GET	/	Public	Serves the Jinja2 game template (index.html)
POST	/score	Public	Submits {score, name} → updates leaderboard, returns {high_score, leaderboard}
GET	/leaderboard	Public	Returns current {high_score, leaderboard} without modification

7. RESULTS AND DISCUSSION

7.1 Functional Testing

Manual functional testing confirmed correct behaviour across the following scenarios:

- Zone progression: Speed and spawn-rate increase correctly at each 600-point threshold up to Zone 10.
- Hitbox accuracy: Duck hitbox correctly allows the pterodactyl at 170 px height to pass overhead; standing hitbox correctly registers the 100 px pterodactyl as a hit.
- Combo system: Consecutive coin collections increment the combo multiplier; taking damage correctly resets combo to zero.
- Lives and invincibility: Three lives decrement correctly; 900 ms invincibility window prevents multiple rapid hits; flicker effect renders correctly.
- Leaderboard: Score submission correctly updates high score when new score exceeds previous best; leaderboard array sorts and trims to ten entries.

7.2 Performance Benchmarks

Test Scenario	Frame Time (ms)	Particles Active	Status
Zone 1, no entities	2.1	0	Pass
Zone 5, car + coin active	3.4	0	Pass
Coin collect burst	3.8	15	Pass
Hit burst + dust puff	4.2	26	Pass
Level-up burst	5.1	30	Pass
Zone 10, all entities + particles	5.9	30	Pass

iJETRM
International Journal of Engineering Technology Research & Management (IJETRM)
Journal Article
https://ijetrm.com/issue/

Score submission (Flask API)	—	—	Pass

All scenarios maintained smooth animation at 60 fps. Frame times remained well within the 16.67 ms budget even at Zone 10 with a full particle burst active. The Flask leaderboard API responded within 40 ms on all local test requests.

8. APPLICATIONS

Wasteland Run is applicable in the following contexts:

- Final year project portfolio demonstrating full-stack web development, canvas rendering, physics simulation, and API design within a single cohesive deliverable.
- Educational demonstration of procedural animation techniques, delta-time physics loops, and particle system architecture for computer graphics or games programming modules.
- Template for browser-based endless runner games requiring no external asset pipeline or game engine, suitable for rapid prototyping or game-jam development.
- Classroom deployment via a shared Flask server, allowing students to compete on a live leaderboard and inspect server-side score management code.
-

9. CONCLUSION

This paper has presented Wasteland Run, a post-apocalyptic browser-based endless runner that combines a Flask score persistence backend with a procedural Canvas 2D rendering frontend. The system demonstrates six-state character animation, three-layer parallax background rendering, AABB collision detection with variable hitboxes, a four-type particle system, delta-time physics, and a REST leaderboard API within approximately 1,400 lines of JavaScript and 60 lines of Python.

The project confirms that a compelling, visually detailed game can be built without external assets, sprite sheets, or game engines, and that Flask provides a suitably lightweight backend for shared score persistence in an academic deployment context.

Future work will pursue five enhancements: (i) mobile touch controls with visual on-screen buttons; (ii) procedural audio using the Web Audio API; (iii) additional enemy types and obstacle patterns in higher zones; (iv) animated level-transition screens between zones; and (v) a persistent user account system replacing the anonymous name entry, with OAuth-based authentication.

REFERENCES

[1] Mozilla Developer Network, "Canvas API," MDN Web Docs, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API

[2] A. Palmas and M. Maggini, "Procedural animation techniques for browser-based games," Int. J. Comput. Graph. Animat., vol. 11, no. 2, pp. 33–44, 2021.

[3] Pallets Projects, Flask Documentation, version 3.0, 2024. [Online]. Available: https://flask.palletsprojects.com/

[4] R. Nystrom, Game Programming Patterns. Genever Benning, 2014. [Online]. Available: https://gameprogrammingpatterns.com/

[5] Google Fonts, "Orbitron," Google Fonts, 2024. [Online]. Available: https://fonts.google.com/specimen/Orbitron