Roguelife

COS426 Final Project - Aetizaz Sameer, Andrew Hwang, Jude Muriithi, Melody Choi

---

Project Repo: https://github.com/jkmuriithi/roguelife426
Project Demo: Google drive link
Note: Demo used special debug features (namely invincibility) to complete the game within the time limit

1. Introduction

Roguelife is a 2.5D roguelike beat-em-up game based on the hardest situations in life. A roguelike is a genre of RPG (Role-playing Game) that is characterized by a dungeon crawl through procedurally generated levels with new scenes to explore, permanent death (if your character dies, you start at the beginning). Procedural generation entails randomly generated levels, providing a new experience with each playthrough. Popular examples of roguelike games are The Binding of Isaac and Hades which were inspirations for Roguelife's design, particularly the latter.

1.1. Goals and Features

Generally, the goal of roguelike games is to progress through each level either by survival or exploration of the map. More specifically, Roguelife aims to provide a unique twist to the classic roguelike game genre by generating various relatable, modern, difficult, yet comedic contexts. The gameplay requires a fair amount of skill developed from playing for longer periods of time which indicates the relatively high difficulty.

Given the time constraints of a final project, our game ultimately included the following:

- One fully developed scenario, with at least one of each of the following Levels:
    - A corridor 'in between' rooms
    - A start room, introducing the scenario and movement mechanics
    - Two fight rooms – one with melee enemies, one with ranged enemies
    - A final boss battle level against a stronger enemy
    - A platformer (parkour) level requiring precise inputs to traverse a level
- Custom game engine created in ThreeJS, with custom memory management for efficient level loading and unloading
- Transparency effects to based on object orientation and player position to ensure that the player and enemies are always visible, even behind objects
- A player with customizable movement mechanics, e.g. double jumping, custom move/jump velocity, and projectile firing
- Enemies of ranged and melee variants, able to damage the player and cause the game to restart from the beginning (in typical roguelike fashion)
- Unique per-level textures, objects, and lighting
- Healthbars, instruction and welcome screens, and plenty of easter eggs

2. Inspiration / Motivation

With this game we attempted to mimic the fundamentals of the roguelike genre (including a room-based level system, permanent death, and movement along a grid. We took heavy inspiration from The Binding of Isaac: Rebirth and Hades for these fundamentals, as well as for the 2.5D view in the case of Hades. We also discussed the potential for Pokémon Black and White-style 3D effect before ultimately settling on the 2.5D view.

- **Clarifying 2.5D**
  In this writeup we describe the game as 2.5D. This refers to a style of game in which there is free camera movement along 2 dimensions, but a third depth dimension is visible on objects. Intuitively, a typical 2.5D game view may appear to be initially from a top-down view (initially giving a 2D view) that is then rotated to reveal a third dimension.
  A 2.5D view can be achieved by rendering objects in 3D (typically using an orthographic camera) and fixing the view accordingly, or in 2D by layering 2D axonometric sprites. For Roguelife we used the former approach, building a 3D application, although we elected to use a perspective camera because it tended to produce better results for large rooms such as the fight and boss rooms.

3. Implementation

3.1. Technology Stack

We used ThreeJS for 3D rendering and Cannon-es for 3D object physics and collision detection. We built the project in TypeScript using Vite as the development server. We found Visual Studio Live Share and Git/GitHub indispensable for collaboration and version control, as well as for hosting a live web demo via GitHub Pages.

3.2. Approach / Methodology

Our implementation of this game involved an OOP-style approach that makes heavy use of class inheritance. Each scene in our game is an instance of a Level object. Additionally, we created classes for many of the objects that would be represented in each Level, including the following:

- Physics Objects (class PhysicsObject), representing any objects subject to the constraints of physics. This allows for interaction between players and objects (poly models) in any given scene.
- Characters (class Character), both playable (Player) and non-playable (Enemy, RangedEnemy, and MeleeEnemy)
- Levels (class Level), representing every 3D scene in the game
- A Level manager (class LevelManager), through which we can manage the trajectory of our game via which level we progress to.

3.3. Physics Objects

Physics Objects are representations of any objects subject to the constraints of Newton's Laws of Motion. The PhysicsObject class acts as an interface between Cannon-es (for

physics simulations) and Three.js (for 3D rendering) which allows for collisions on the 3D models that we use. Each PhysicsObject is able to have configurable properties such as names, shadows, opacity, and collision shapes while also associating a Cannon-es Body with the object. Helper methods aid in setting, updating, and resetting the body's position when creating level scenes. In each frame, positions are calculated using the Cannon-es Body type, and the resulting positions are copied into the ThreeJS object. In short, this class acts as a bridge between Cannon-es physics simulation and Three.js rendering, allowing for synchronization between the two physics-based interactions in a 3D environment.

3.5. Characters

Characters are a broad subclass of PhysicsObject that further implement projectile configurations (see previous section) and healthbars, and default to rectangular prism shapes.

3.5.1. Player

Player is a subclass of Character that further implements an event listener for keystrokes, movement along the grid, and (double) jumping. Each level generally contains a player controlled by the user using WASD/arrow keys for movement, spacebar to jump, and return/enter to fire a projectile.

3.5.2. Enemy

Enemy is a subclass of Character that further implements contact damage and stores player position for pathfinding/targeting purposes.

3.5.3. Melee Enemy

MeleeEnemy is a subclass of Enemy that, rather than remaining still, seeks out the player to damage them via contact damage.

3.5.4. Ranged Enemy

RangedEnemy is a subclass of Enemy that, rather than remaining still, attempts to remain around its spawn location, moving back and forth to avoid player projectiles. It also moves back toward its spawn location if moved via knockback or being pushed by the player or another enemy. It also has a projectile configuration and fires projectiles in the player's direction, at a rate specified by the developer for each individual RangedEnemy.

3.5.6 Projectiles

Each Character has a boolean instance variable firedProjectile which indicates whether or not the Character has called its fireProjectile in the given frame. If the flag is set to true in a given frame, then a projectile will be asynchronously generated and added to the level with the appropriate position, orientation, and velocity.

3.6. Level Components

3.6.1 Wall
Wall is a subclass of PhysicsObject that uses ThreeJS BoxGeometry to create a box (rectangular prism) in 3D. By default it is used for walls, floors, ceilings, and platforms – with a small thickness to mitigate PhysicsObject clipping – but it can also be used as a general-purpose physics-enabled box with a larger thickness. It inherits physics properties from PhysicsObject and creates a mesh using the box geometry and material parameters to define its appearance. We are able to customize properties of the wall such as size, color, and opacity settings.

3.6.2 Room
Room is a subclass using the Three.js library to create a 3D room with walls. A Room is constructed by creating a customizable rectangular prism of six walls ('floor', 'leftBackWall', 'rightBackWall', leftFrontWall', 'rightBackWall', and 'ceiling') which are all instances of Wall. Properties of Room can be customized to fit the various interactions associated with level design.

3.6.3. Level
Level is our manner of organizing all the components that are added to a Scene for ThreeJS to render. The Level class keeps track of the room it is based in as well as a player (optionally), enemies (optionally), 3D physics objects, lighting, active projectiles, and a 'portal' which leads to the next room after all enemies have been defeated.

3.6.4. Level Manager
LevelManager is a class that stores each level of the game and handles the logic of transitions between levels, moving the game forward and backward when the player finishes a level or is defeated. For the purpose of preserving video memory and ensuring smooth gameplay, rather than loading every level at once, LevelManager only loads a given level as the player enters it, freeing associated memory once the player has completed the level. In order to show text on-screen to the player, we temporarily switch out the LevelManager's current level with a ThreeJS Scene which contains a single 2D CSS object.

3.6.5. Polygon Models
Our group used ~40 polygon models from the website poly.pizza, particularly public domain models and models that were part of the Google Poly project (now shut down). Non-public domain models are cited in the file credits.txt.

3.6.5.1. Dynamic Opacity
PhysicsObjects have a configuration parameter opacityConfig which can be used to specify whether or not the object's opacity changes dynamically based on the position of

the player and the angle of the camera. There are two strategies that we use to determine the dynamic opacity of such objects, which we call the “directional” and “character intersection” strategies. If an object has directional dynamic opacity, then it must specify an “opacity normal” vector. If the dot product of this opacity normal and the camera’s direction vector is above a certain threshold, then the object will be set to a lower opacity. If an object has character intersection dynamic opacity, then we’ll set it to a lower opacity anytime its bounding box intersects any ray between the camera and a character in the level.

3.6.6. Textures

Our group used ~25 textures for characters, carpets, logos, walls, etc. Most were images from the internet used for atmosphere and setting, whereas a few were made by hand in Photoshop and used for player and boss textures. Textures were implemented via ThreeJS MeshPhongMaterials, which can take a texture map to assign to a mesh object. For some objects (such as standard enemies) we elected to use colors rather than textures for visibility. Non-public domain textures are cited in the file credits.txt.

3.6.7. Lighting

Each level in the game has a corresponding <LevelName>Lights class which contains its ThreeJS lights. In general, we use a main hemisphere light along with a combination of shadow-casting spotlights to light each level. We use our spotlights to highlight the detail of our level design and greatly enhance the appearance of the game overall. We mitigate the performance demand of having several shadow-casting lights in each level by reducing the resolution of each light’s shadow map, if necessary.

3.7. Game Design

Roguelife’s first setting is modeled after a generic big tech company. The interior design reflects that of a corporate tech giant with an abundance of desks and cubicles. The walls are a unique distinguisher to finance companies as they generally don’t use colorful wall designs. In this setting the player goes into the company building in order to interview for a job at the company. Will the player be able to get hired?

3.7.1. Reception Desk

The Reception Desk is the first level that the player spawns in. The room is decorated with bright colors and lively decorations that reflect a generic Reception Desk. All the items you see are PhysicsObjects so they can be interacted with and moved around. Following a brief introduction of the controls, the player is met with a Receptionist NPC and the first Enemy. Defeat that enemy in order to proceed to the next level.

3.7.2. Corridors

The Corridors are connecting levels that lead the player to the next level. Every corridor is furnished like a corporate floor with cubicles, desks, chairs, computers, a printer, and a

water dispenser. Each time the player enters a corridor, the camera angle and lighting will randomly change, thereby giving the player the sensation that they have entered a new and different corridor.

3.7.3. Fights

The Fight levels are levels in which players must defeat a set of Enemies to proceed with the game. The first Fight level is an expanded corporate floor featuring a group of Melee Enemies that rush towards the player after being interrupted from their work. Each cubicle and desk is an immovable physics object to maintain the structure of the map and hopefully guides the player to navigate safely. The second Fight level features a group of Ranged Enemies that are equipped with paper airplane projectiles that the player also uses. In this level, chairs and various objects are movable physics objects which shield the player from incoming projectiles – but also shield enemies from outgoing projectiles. Fight your way through both of these Fight levels in order to proceed with the story.

3.7.4. Boss

The Boss level is the final level that the player must clear in order to beat Roguelife. The Boss level features two Melee Enemies, two Ranged Enemies, and the Boss! The Boss is extremely powerful with a large amount of health and huge, high-damage projectiles. The player's goal is to skip the entry level jobs and get hired as the CEO, taking out all the enemies that stand in your way. The Melee Enemies charge forward toward the player on the ground floor while the Ranged Enemies rain down projectiles from above on elevated platforms. The player must first get rid of the Melee Enemies and then perform some parkour to reach the Ranged Enemies on those platforms. Finally, the Boss is all that stands between you and victory. Defeat the Boss for a special prize at the end. The back wall features a funny Easter Egg!

4. Discussion

Our goal for Roguelife was to challenge us in the most applicable way possible: create the best game we can with the resources we learned from the semester, starting (nearly) from scratch by making our own game engine. Although our approach and design for the game was extremely well thought out, it would've been nice if we knew that the vision we had for the game was a little too ambitious for a final project that was meant to be done within 2 weeks.

We gained many skills from creating Roguelife. Notably, the OOP design greatly improved the efficiency of level building and scene generation by allowing us to abstract away the redundancies. However, we didn't know that level design would be as much work as we had expected. This was by far the most time-consuming part of the project, and in retrospect it likely would have been more time-efficient to implement a basic level builder to speed up the process. We had several bugs which were time-consuming to fix, such as improper orientation of projectiles (which we fixed with trigonometry) and 3D models

being misaligned with their world coordinates due to inaccuracies in the underlying assets (which we fixed by centering and rotating the models wherever they were used).

Contributions

Aetizaz:
I worked on most of the major sections of the game. I designed the second Fight level and implemented Enemies, and also worked on the game engine (Level and physics that initially were mostly in Player and Wall/Room and eventually moved to PhysicsObject) with Jude. I also implemented our first version of textures and designed some textures (e.g. for the player, boss, office background, and boss battle background) in Photoshop.

Andrew:
I collaborated with Melody and Aetizaz on Level design and lighting. I designed and implemented one of the Fight Levels and the Boss Level through texture mapping and poly model imports.

Jude:
I mainly worked on the foundations of our game engine, implementing dynamic opacity and most other key features of Level, LevelManager, PhysicsObject, and Player. Throughout the project, I wrote engine helper functions as the rest of the team needed them, and I also reviewed the game code often to simplify and optimize the codebase over time.

Melody:
I mainly worked on Level design, importing and working with poly models and various textures to make the interactive scenes of the game fun and visually appealing. I designed the Office Reception and Corridor levels, which served as a template for other levels in the game, and retextured assets to fit with the theme of the game.

Works Cited

- ThreeJS
- Cannon-es
- The Binding of Isaac: Rebirth
- Hades
- Pokemon Black and White
- 3D models from poly.pizza and the former Google Poly project, cited in credits.txt
- Various tutorials and documentation pages cited in the code directly
- Various textures cited in credits.txt