CS462 Final Submission

Departed Affairs and Co.

By Calvin Mickelson & Hayden Decker

Introduction

Our ideal game we wanted to make is a roguelite action adventure set in the early 1900s. The player controls Dr. Vibble, a traveling spiritualist hired by the living to cross into the spirit world and resolve unfinished business for the dead. Each mission takes place on a procedurally generated map filled with enemies, hazards, story events, and upgrades, with the long term goal of growing strong enough to defeat powerful boss spirits.

The game is split into two parts: the overworld and the underworld. The overworld is the player's hub where they rest between runs, visit shops, manage equipment, and pick new assignments. Jobs are chosen in the cemetery by interacting with marked gravestones, each tied to a specific spirit.

The underworld is where the main gameplay happens. Levels contain a set of procedurally generated rooms connected by shifting paths, populated with hostile spirits, helpful NPC spirits, side quests, and occasional boss encounters. Bosses are corrupted versions of the target spirit, far stronger and equipped with unique attacks. After completing an assignment, the player returns to the overworld to collect their reward and prepare for the next mission.

Technical Details

For a full visual overview of the engine and its processes, see the flow chart at this link.

Content Loading Pipeline

Manifest, Storage Roots, and Cache Layers

Manifest project layout

We have a single manifest.json instead of baking content into the build. When the engine starts, it checks that the folders under project/SRC exist so assets, loading screens, and shared content load without extra setup.

Storage and cache

Uploaded PNG files stay in SRC/asset. Prebuilt textures and light maps sit in cache/ so the renderer can reuse them instead of reprocessing images every launch. asset_tool.py and light_tool.py rebuild the cache when the manifest flags a change, and the rebuild queue runs those tools before the game begins.

Asset Library and Texture Pooling

AssetInfo metadata and shared caching

The AssetLibrary keeps the manifest and SRC/assets folders in sync and accessible to individual asset instances. It builds AssetInfo records for animations, lights, spawn rules, and texture paths. Assets use that shared data so new spawns do not reread JSON. Once rendering is ready, the library or the map loader preloads animation variants so GPU-ready frames are waiting before assets enter the world grid.

Python-powered texture pipelines

asset_tool.py reads the manifest, looks at source frames in SRC/assets/<asset>, scales them to the needed sizes, applies simple effects, and writes ready PNGs into cache/<asset>. light_tool.py rebuilds light masks when the manifest marks them stale. Because these tools follow the manifest, the game only reads prepared files and never spends frame time on heavy image work.

Map, Room, and Trail Construction

Manifest map layers, rooms, and trails

Each map entry in the manifest lists layers, room data, trails, boundaries, assets, and grid settings. The loader reads those values, figures out the map size, and fills in spawn controls. If a required layer is missing it creates a simple fallback room so there is always somewhere to play.

Room geometry, spawn groups, and trails

The room generator turns layer data into rooms with shapes, spawn groups, and rules on how children inherit properties. The spawn planner samples rooms and trails, respects spacing filters, and finalizes assets once placed. Trails use the same system but focus on path-shaped areas to control where things appear.

Grid settings and Caching

Map grid settings give the resolution and chunk sizes. The loader uses them to build the world grid and to tune how assets handle collision, depth sorting, and camera culling so the layout stays consistent across rooms.

Scene & Asset Structure

World Grid, Chunks, and Rooms

The world grid keeps assets in chunks and points so collision and visibility checks stay fast. Chunks appear when needed and assets register themselves at exact points. Only the camera's view stays active.

Asset metadata, child assets, and tiling

Each asset carries shared info, a spawn id, animation state, and optional tiling data. The shared info lists lights, shadows, child assets, and spawn groups. Child assets use animation timelines, and tiling lets a single asset repeat across a larger area without creating many copies.

Animation runtime and custom controllers

The animation runtime steps frames, moves assets, and works with controllers. The update layer adds helpers for pathing, movement, and animation tweaks. Controllers come from manifest keys and can be built-in or custom.

Runtime Flow & Rendering

Main loop, input, and asset updates

The main loop runs at 24 FPS. It reads input, sends events to assets and the player controller, then updates each asset. Updates run controller logic, advance animations, and flag render data when visuals or lighting change. During setup the app also starts audio and drops into Dev Mode if no player asset is present.

Screen grid, camera, and lighting

The warped screen grid keeps the camera aligned with the map and smooths zoom and pan. It rebuilds every frame from the world grid, so the map-to-screen link stays right. Camera and lighting toggles feed both gameplay and Dev Mode, while light updates go through the light map tools that serve the renderer and editor panels.

Render pipeline and composite packages

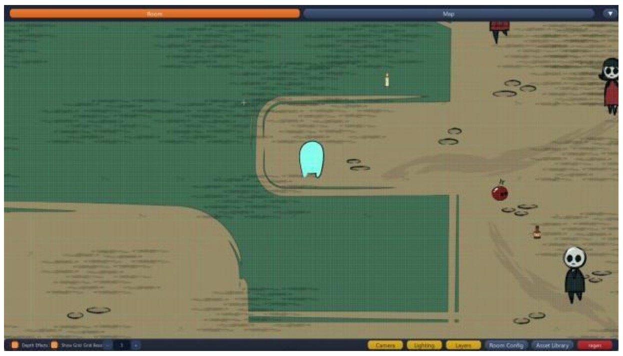
The scene renderer clears the screen, draws background layers, and asks the composite renderer to build each asset's render package. Packages gather textures, child attachments, and lighting in order. After that, the renderer sorts by depth, draws lighting, and adds depth cues, dark masks, and any post effects.

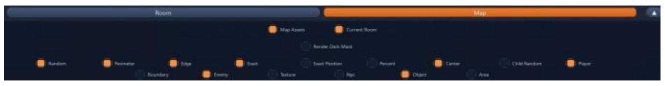
Dev Mode

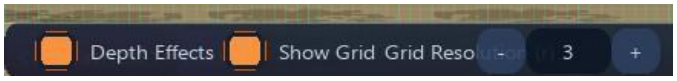
Dev Mode turns the engine into a live editor. It prioritizes responsive edits over polish, saves changes back to the manifest/SRC assets, and can be opened with Ctrl+D or by loading a map without a player. Layout preferences live in dev_mode_settings.json so teams share the same workspace defaults.

Main screen and navigation

- World view stays live while editor UI floats above it for room, map, and lighting tasks.
- Header toggles Room/Map/Lighting modes; quick filters limit which assets render while you edit.
- Footer carries notifications, quick tasks, camera shortcuts, and grid/depth toggles with adjustable resolution.

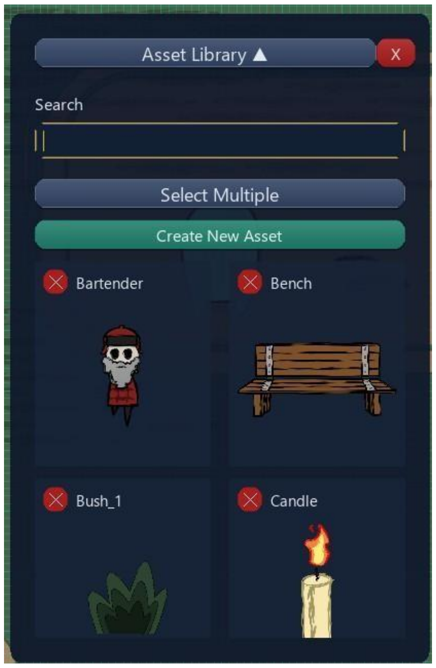






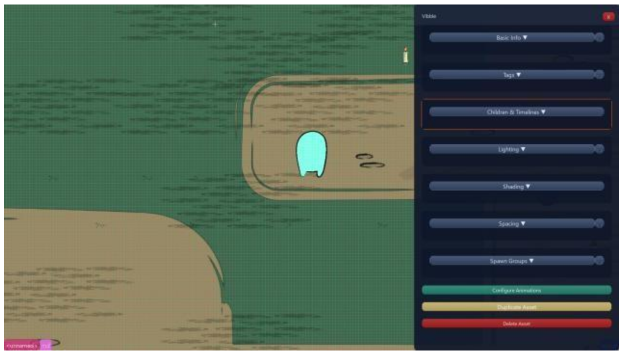
Asset library panel

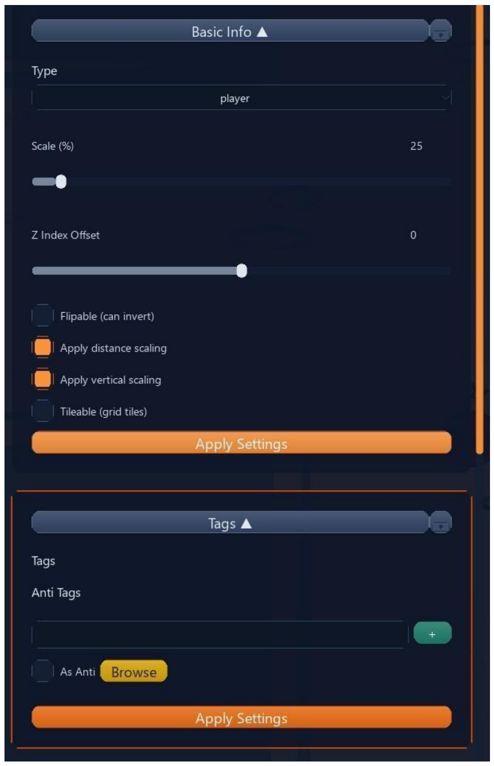
- Search, multi-select, and drag assets from the manifest directly into the world.
- Use “Create New Asset” to spin up a fresh asset_info_ui entry; right-click to open details for existing assets.
- Delete assets from the manifest via the red corner button when cleaning unused content.



Asset info overview and basics

- Overview groups basic info, tags, children & timelines, lighting, shading, spacing, and spawn groups; Apply writes back to manifest/SRC.
- Set type, scale, Z offset, flip/tiling flags, and start animation to match controller expectations.
- Tags and anti-tags drive spawn rules and filtering so runtime and editor stay consistent.





Type: player

Scale (%) 25

Z Index Offset 0

Flipable (can invert)

☑ Apply distance scaling

☑ Apply vertical scaling

Tileable (grid tiles)

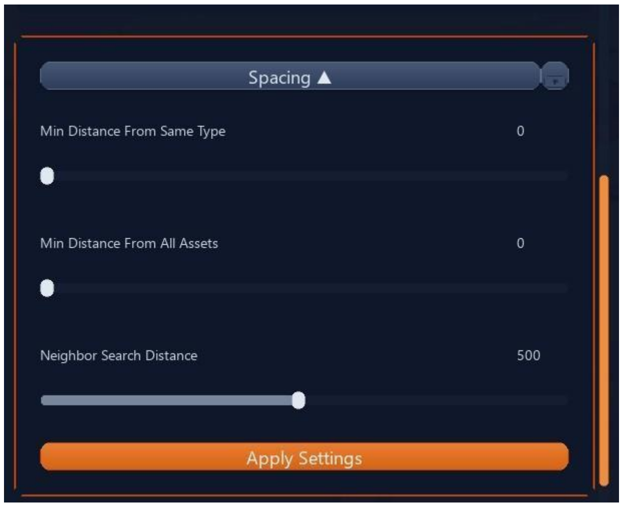
Tags

Anti Tags

As Anti

Spacing controls

- Minimum distance per type and global distance prevent dense spawns from overlapping.
- Neighbor search distance tunes how far the spacing check reaches when scattering assets.
- Apply settings writes spacing rules into the asset definition for future drops.



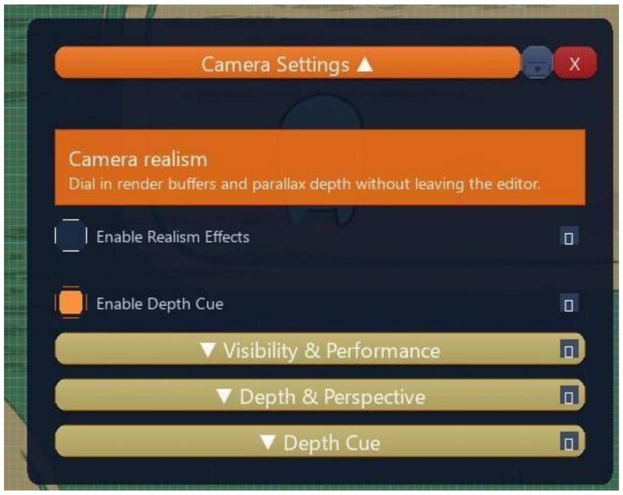
Min Distance From Same Type 0

Min Distance From All Assets 0

Neighbor Search Distance 500

Camera settings

- Toggle realism effects and depth cue while adjusting render buffers without leaving the editor. (Realism is disconnected from current engine version ☹)
- Visibility & Performance plus Depth & Perspective panels balance clarity with cost.
- Depth cue options help align parallax and focus when testing camera moves. (Disconnected from current engine version)



Camera realism

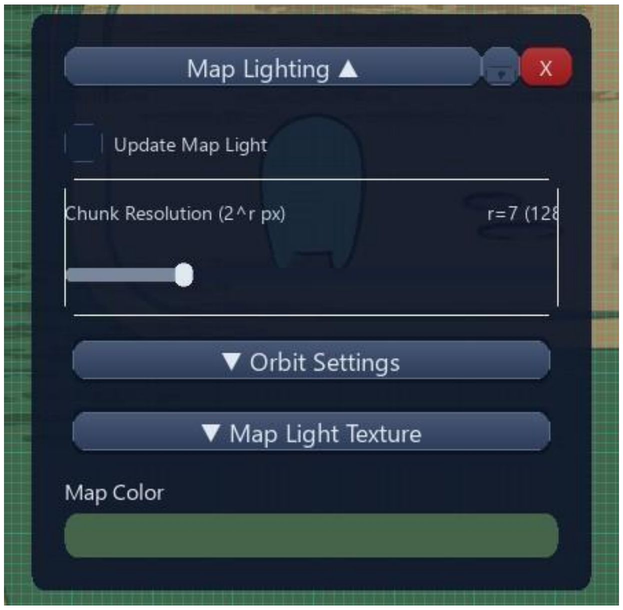
Dial in render buffers and parallax depth without leaving the editor.

Enable Realism Effects

Enable Depth Cue

Map lighting panel

- Update map light toggles, chunk resolution, and orbit settings before rebuilding light maps. (Disconnected logic)
- Texture and color controls preview how lighting combines with the current grid resolution.
- Changes flag light_tool.py for rebuild so renderer and editor stay in sync.



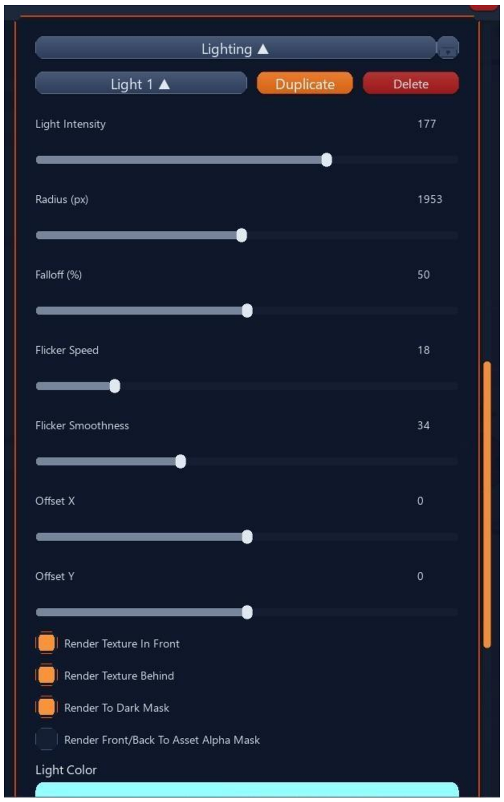
Update Map Light

Chunk Resolution (2^r px) r=7 (128

Map Color

Map-mode lighting editor

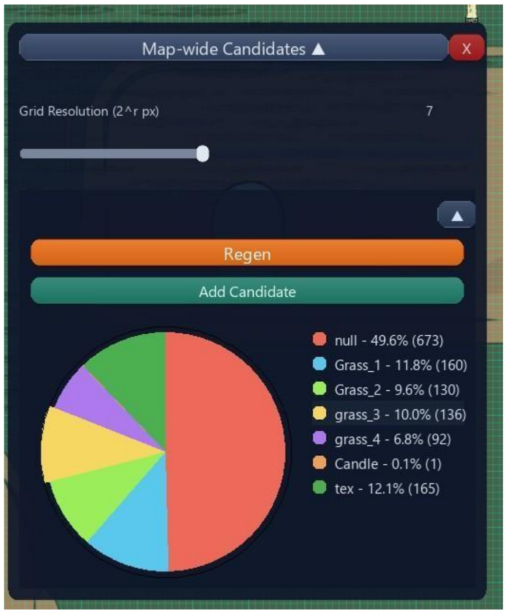
- Per-light controls adjust intensity, radius, falloff, flicker speed/smoothness, and offsets.
- Duplicate/delete light slots to try variants without touching scene data. (Disconnected logic)
- Render toggles handle front/back texture masks to match desired depth ordering. (Disconnected logic)



Light Intensity 177
Radius (px) 1953
Falloff (%) 50
Flicker Speed 18
Flicker Smoothness 34
Offset X 0
Offset Y 0
[✓] Render Texture In Front
[✓] Render Texture Behind
[✓] Render To Dark Mask
[ ] Render Front/Back To Asset Alpha Mask
Light Color

Map-wide asset candidates

- Set grid resolution for map-wide scattering and regenerate distributions with one click.
- Add or remove candidates; pie chart previews spawn weights before committing.
- Use this view to validate biome mixes or clean up overrepresented props.

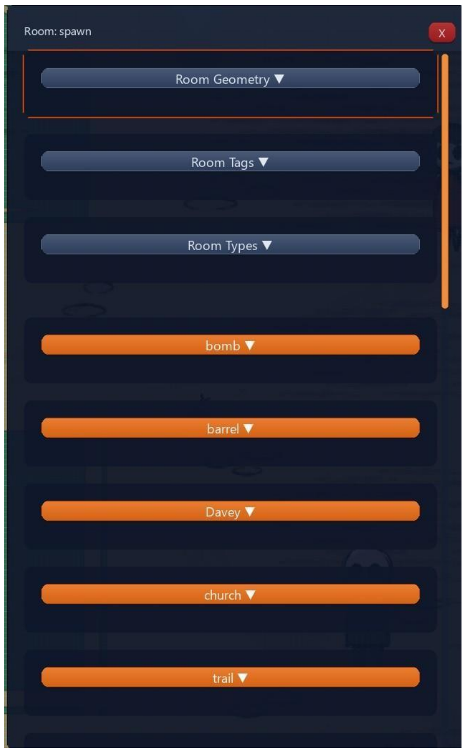


Grid Resolution (2^r px) 7

● null - 49.6% (673)
● Grass_1 - 11.8% (160)
● Grass_2 - 9.6% (130)
● grass_3 - 10.0% (136)
● grass_4 - 6.8% (92)
● Candle - 0.1% (1)
● tex - 12.1% (165)

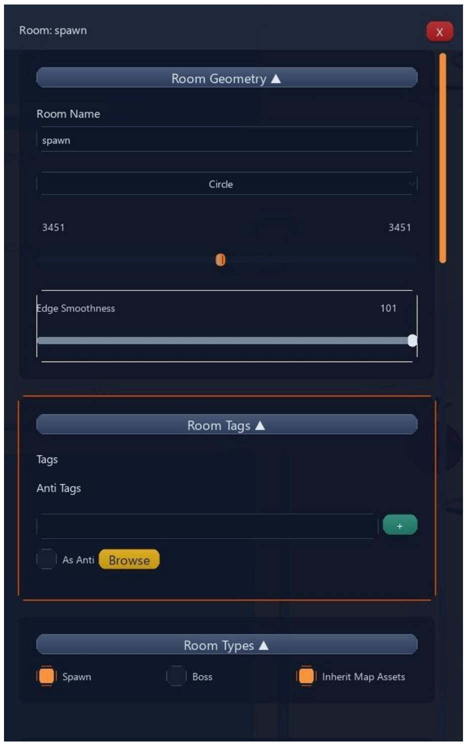
Room editor (collapsed/quick view)

- Collapsed view keeps geometry, tags, and type lists accessible while leaving the play area clear.
- Use it to jump between rooms and spot-check settings without expanding full panels.



Room configuration (expanded)

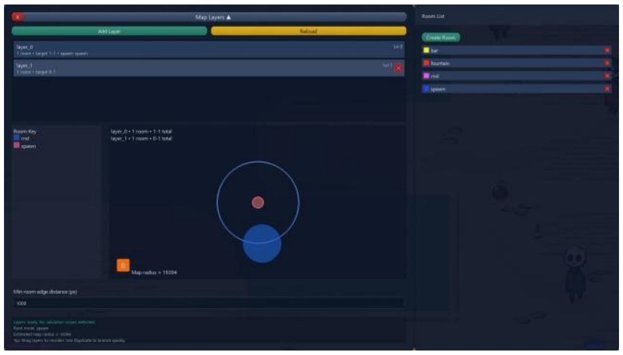
- Edit room name, shape (circle/other presets), radii, and edge smoothness for layout testing.
- Assign tags/anti-tags and room types (spawn, boss, inherit map assets) to drive procedural rules.
- Expanded view helps verify metadata stays aligned with manifest expectations.



Room Name
spawn
Circle
3451 3451
Edge Smoothness 101
Tags
Anti Tags
As Anti
Spawn ✓ Boss Inherit Map Assets ✓

Room layers view

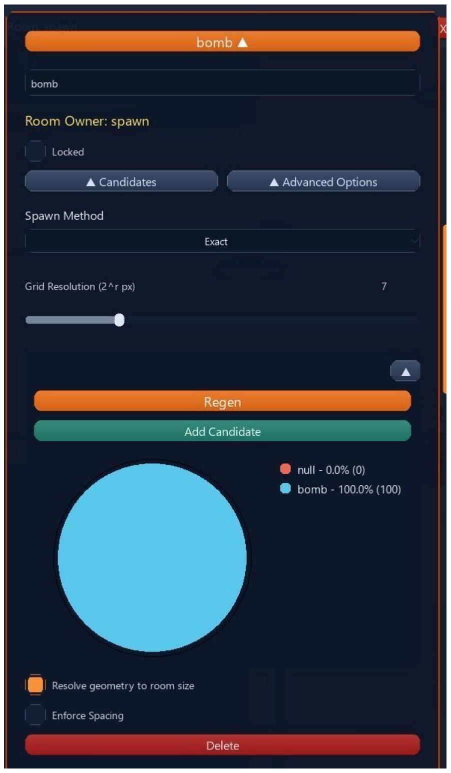
- Stack layers with target counts, reorder, duplicate, and reload to quickly branch layouts.
- Room list shows available templates; the map canvas validates radius and layer separation.
- Layer and room keys clarify which group is active while editing spawn layouts.



layer_0
1 room • target 1-1 • spawn: spawn
layer_1
1 room • target 0-1
Room Key
🟦 mid
🟪 spawn
layer_0 • 1 room • 1-1 total
layer_1 • 1 room • 0-1 total
Map radius = 10394
Min room edge distance (px)
1000
Layers ready. No validation issues detected.
Root room: spawn
Estimated map radius = 10394
Tip: Drag layers to reorder. Use Duplicate to branch quickly.
🟨 bar
🟥 fountain
🟪 mid
🟦 spawn

Spawn group configuration

- Bind spawn groups to a room owner, choose spawn method, and set grid resolution.
- Regenerate and tweak candidates; pie chart confirms weights before saving.
- Options to resolve geometry to room size and enforce spacing keep drops predictable.



bomb

Room Owner: spawn

Locked

Spawn Method

Exact

Grid Resolution (2^r px) 7

● null - 0.0% (0)

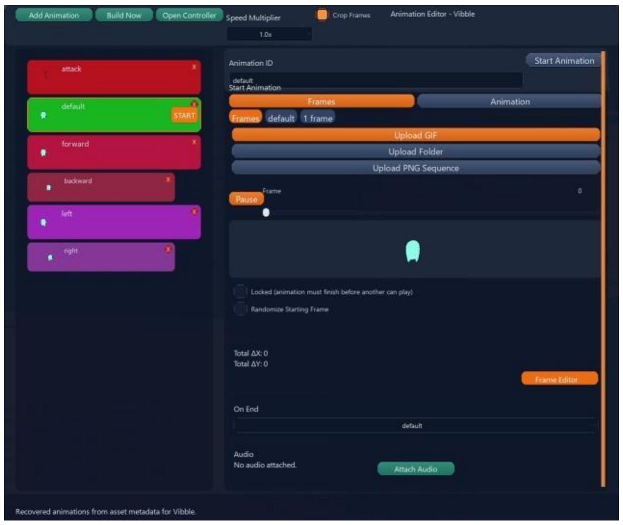
● bomb - 100.0% (100)

[✓ Resolve geometry to room size]

Enforce Spacing

Animation editor

- List of animations per asset with add/build controls and a start animation selector.
- Upload GIFs, folders, or PNG sequences; crop frames and preview playback speed.
- Lock or randomize starting frames, attach audio, and jump into the frame editor.



Speed Multiplier: 1.0x

Animation ID: default

Start Animation

Frame 0

Locked (animation must finish before another can play)

Randomize Starting Frame

Total ΔX: 0
Total ΔY: 0

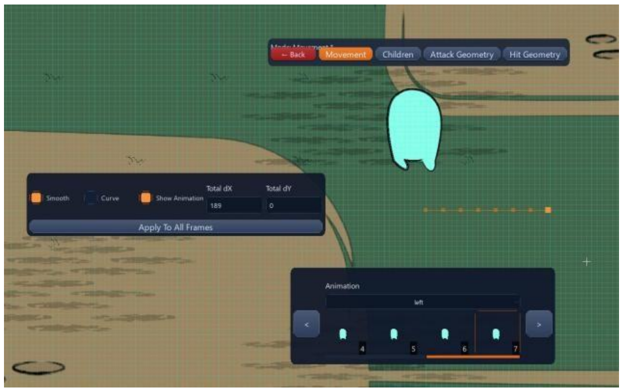
On End: default

Audio
No audio attached.

Recovered animations from asset metadata for Vibble.

Frame editor: movement

- Movement tab sets total dX/dY per frame and can apply to all frames at once.
- Smooth/curve toggles and show animation overlay aid path tuning against the grid.
- Navigation bar swaps between Movement, Children, Attack Geometry, and Hit Geometry modes.

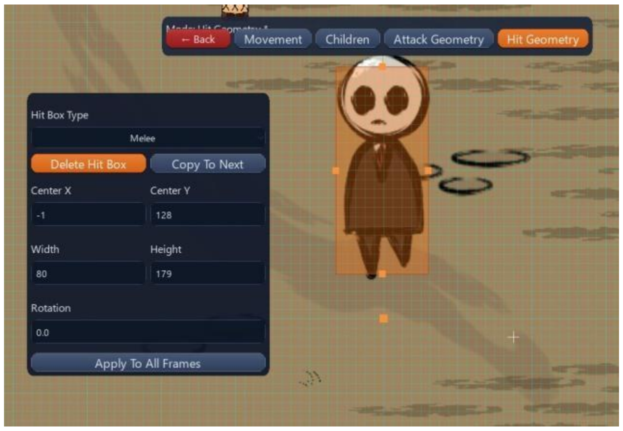


Total dX: 189
Total dY: 0

Animation: left

Frame editor: hitbox panel

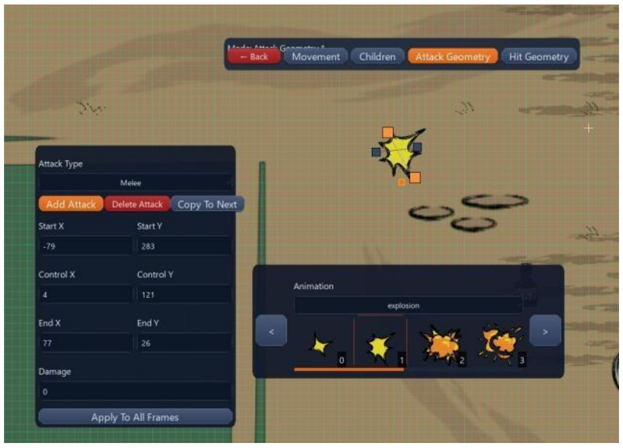
- Define hit box type, center offsets, width/height, and rotation for the current frame.
- Copy to next frame or apply to all frames to speed collision authoring.
- Visual overlay keeps the hit volume aligned to sprites while adjusting numbers.



Hit Box Type
Melee
Center X
-1
Center Y
128
Width
80
Height
179
Rotation
0.0

Frame editor: attack geometry

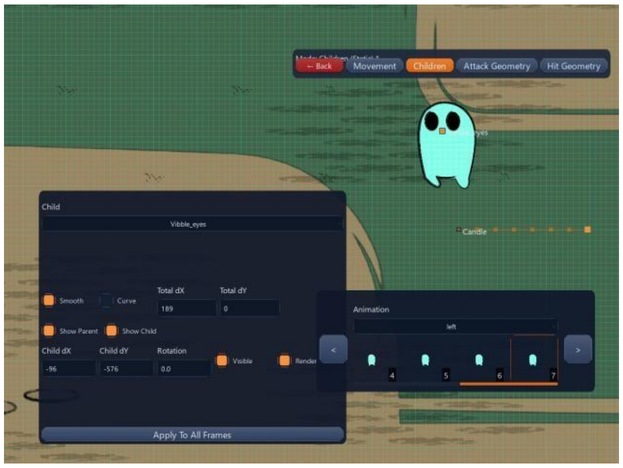
- Author melee attack paths via start/control/end points and damage values.
- Copy settings to the next frame or apply globally to keep combos consistent.
- Animation picker previews the attack FX frames alongside control points.



Attack Type
Melee
Start X
-79
Start Y
283
Control X
4
Control Y
121
End X
77
End Y
26
Damage
0
Animation
explosion

Frame editor: children and attachments

- Attach child assets (e.g., eyes, effects) with per-frame offsets, rotation, and visibility.
- Toggle parent/child visibility while aligning attachments to the main animation.
- Apply to all frames to keep attachments synced across the timeline.



Child
Vibble_eyes
Total dX
189
Total dY
0
Child dX
-96
Child dY
-576
Rotation
0.0
Animation
left

Game Challenge

The main challenge we chose for our game was to create it from a custom engine. Major engines, such as Unity which we were taught for the class, are usually designed to be general purpose. We wanted to come up with an engine designed entirely for 2dimensional games. With this, we could remove a lot of the bloat that takes up processing power within other game engines. Overall, this added a huge amount of work to the game. Not only did we have to think of designing the game, we also needed to design an engine from the ground up. Some of the biggest challenges when creating this were issues with: The camera and depth system including parallax and realism, keeping geometry consistent across the map grid, keeping JSON data, in-game assets, and editor state in sync during runtime, handling legacy data as features were added, as well as data and class ownership hierarchy. Throughout the course of making the engine, the overall design was changed many times to try and increase overall performance as well as simplify the original design.

The other challenge we wanted to try and tackle during this project was getting combat working. While we have working assets with their own animations, we didn't have a great way for players to attack enemies at the start. When working with Unity, you can easily call an object's collides function to then handle all of the details of attacking. For our engine, this is something we had to design from the ground up. Our way of dealing with this was to create child animations that an asset has ownership over. When an asset attacks another asset, we could then use this child to calculate if the animation overlaps with the other entity, if it does we could then apply the attacking logic on the asset. This gave us tons of trouble within the final few weeks of the project.

Along with these challenges we decided to tackle, we wanted to have a few quality of life features to help us in our development cycle. We can call these extra challenges as well that we worked on. The first one was an integrated room editor built directly into the engine. Within this room editor, the user can select rooms, view the bounds of rooms, place and drag spawn group configurations, and lastly configure asset animations. The second feature that was added was procedurally generated rooms. This works through taking data from JSON objects to randomly configure and regenerate rooms instead of having to hand-edit data. The last feature that we added was a UI panel for adjusting camera settings. Within this UI, the user can change realism settings, zoom settings to focus on key points, and depth cue controls.

Gameplay / Game Information

The genre of the game we wanted to create is a roguelike. Roguelikes, the overarching genre, is a style of role-playing game where generally the main character must battle through dungeons to the end. The world is seen from a bird's eye view moving through a grid-based movement system and plays in a turn based style. If the player dies then the character is permanently dead. Roguelites are a sub-genre, characterized by having a procedurally generated world, persistent upgrades, and semi-permanent death. In some cases the death can restart the player's progress to zero and in other cases it only resets a player's gear and abilities. Some examples include D&D, Hades and Hades 2, slay the spire, and Spelunky.

Evaluation

We did not have an evaluation conducted on our project. Due to this we were unable to add any feedback.

Member Responsibilities

Overall engine work was done by Calvin before as well as during class. Things such as the multiple UI's, lighting, and map work were all created by Calvin. Throughout the semester, Hayden helped to work on a few minor things in the engine, attacking logic, animations, and general design planning.

Conclusion

Departed Affairs & Co pairs a custom-built 2D engine with procedural roguelite gameplay and a toolkit that brings the overworld hub, cemetery job selector, and underworld missions together in a cohesive way. The overview above connects those narrative ideas to the technical systems that make the experience playable, extendable, and ready for future improvement.

We worked extremely hard on this project. We wanted to build the game we envisioned, and we underestimated how much work that would take. The gameplay isn't where we hoped it would be, even though we pushed hard to get it there. Dev mode took far longer than expected, lighting and scaling caused constant issues, and refactoring the grid system for better attack logic created new problems we are still solving.

Even with the shortcomings, we are proud of what we accomplished. With a little more time, we truly believe we could have reached the full vision. We plan to keep working on it and eventually get the game to where we always wanted it to be.