Final Submission: Lamp

Introduction

I have created my game with a data-driven, highly extensible approach that makes it somewhat closer to a game engine than a game. The game I created is akin to a horror-like metroidvania controlled like a twin stick shooter.

Demonstration video: https://youtu.be/_hHLO5_hr88

Game screenshots can be seen in the Guide.

Context

This project draws on a lot of different games and genres for its mechanics, and adds some novel variations on top of them. The basic controls of the game make the core gameplay feel akin to a dungeon-crawler, or games like "The Binding of Isaac", exploring a series of rooms to find items that enhance your abilities, and ultimately the exit. However, these games are often focused on RPG-like progression and direct combat with enemies, while this project draws upon horror games for this. The enemies in the game cannot be defeated, only slowed down, akin to "Mr. X" from "Resident Evil 2", whom you spend a lot of that game just trying to escape from, or the enemies in "Outlast". This project also uses a darkness mechanic, with a limited light supply, a mechanic used in a lot of horror (and many non-horror) games (e.g. Minecraft), though I couldn't find any popular examples of this being done in top-down 2D. Additionally, many dungeon-crawlers make use of some sort of random generation (especially rogue-likes such as "The Binding of Isaac), while I chose to have designed levels so that I could add Metroidvania-style progression, with interconnected areas and using new abilities to unlock new paths. Similar to rogue-likes, however, you must restart the game from the beginning on death. I may change this if the game was larger, but given progression is more through learning the layout of the map and what the items do, I think this is a suitable gameplay decision that also adds to the tension created by the horror-like elements.

There are two games in particular I took direct inspiration for game mechanics from. First is Eternal Darkness, which has a "Sanity Meter", which causes a lot of effects to mess with your head, such as applying distortions to the screen or having illusory enemies appear. Second is Alan Wake, a horror game in which you can can stun enemies with the light from a torch. In that game, it made a lot of the enemies trivial as you could also defeat them with a gun, so I expanded on this by requiring the player to keep the enemies in the light for a significant time

in order to stun them, and having no way to defeat the enemies (meaning this mechanic only slows them down). I also took some inspiration from the classic Doom in how collision between entities is handled (using a grid to optimise it), and the idea of showing some sort of player status through a graphic of the character's face. In Doom, this showed how damaged the character is, but in my game a conventional health bar is used for this, and the face is used to give a rough guide to the status of the "sanity meter".

Notable Features

The environment has dynamic lighting, which allows for multiple light sources (currently only the light from the lamp, and a little glow around the player so you can still see when the lamp is off). In the main menu you can choose between smooth (gourad) shading, and flat shading.

Multiple difficulty modes, chosen from the main menu - different difficulty modes apply a set of multipliers to things such as the enemy speed and damage. These are not all scaled the same, so the different difficulties have a different feel beyond just numerical difficulty.

The map, tiles, enemies and items are all loaded from data files. (there is a helper script which generates the map JSON from an image) Other data in the game is structured in such a way that it could easily be loaded from a data file as well, such as player data and the difficulty modes.

Enemy behaviour is controlled through a FSM in their data, allowing a lot of control in the design of how each enemy behaves without writing any extra code.

Enemy pathfinding uses A* search.

Grid-based world representation which creates a tile-based environment and optimises collision

Camera-based view, which smoothly accelerates to follow the player, hanging ahead of the player's facing direction to give a better field of view. The camera also has a zoom function, which is used to reduce the field of view while the lamp is off to add to the tension created by the darkness.

A madness mechanic, which creates an alternate failure state and adds to the horror elements by creating effects the higher your madness (currently just a screen ripple courtesy of an example shader by Adam Ferriss)

Dynamic tile-based environment, which allows data-drive environment interaction through traversal types, and damage states which allow the player to affect the environment to create new paths.

Multiple routes through the game, including many secrets and secret areas

The game has sound effects and a soundtrack, including title music, all made using FamiTracker. (winning the game being the only event accompanied by silence is intentional)

Design & Implementation

Principles

One of the core design principles I had in this project is to stick to a more functional, data-driven approach, rather than the object-oriented approach I took with previous projects. While OOP concepts such as inheritance do make some programming tasks neater, actually building thos objects leads to code I would consider quite bloated and messy, with lots of boiler-plate and structural code. Javascript makes it quite easy to program in a data-oriented, more functional way. For example, JSON files can be loaded directly into objects which match the structure of the JSON, and objects can contain functions and references to functions as attributes, allowing me to create function tables. Instead of calling the method of an object, I can use data from the object to select a function from the table in much the same manner, and can pass the function the data as an object, resulting in functions similar to those belonging to objects in OOP, but referencing an argument rather than `this`.

Two other core principles I had, to help my produce cleaner code, was to write many, smaller functions, and to completely separate functions for updating game log, and functions for drawing things. This separation also makes it easy to do things such as have the game pause while the inventory is open, but still have the game drawn behind the inventory screen.

The World

Much of this code was taken from the previous project, where the world is represented as a grid, and collision with the environment being handled through properties of each cell of the grid (i.e. whether or not that cell is a wall), and the which entities to use for collision is based on maintained lists in each cell of which entities are in them. When adapting it for this project, I also fixed the very messy code from the previous project used for detecting when an entity enters / leaves cells (this messy code being the reason for the game-breaking bugs in the last project) - now the `updatePos` function, which also stops entities clipping through walls.

Since the world map is much larger than one screen for this game, I extended this to load a world from data (rather than the random generation used in the last project), and use a "camera" to select a subset of the world grid to be the `onScreenWorld`. For convenience and optimisation, this is stored as a separate object which is updated when the game detects that the cells on screen have changed. This object is used for entity collision, and for lighting, by iterating through the cells in this object.

I also extended the simple binary state of being either a wall or not a wall into a tile system. Each cell in the world has a tile, which defines the appearance and behaviour of the cell. Whether an entity can traverse a cell is defined by the tile's `travType`, and whether you can enemies can see through a cell or light can pass through it is defined separately by an opacity

boolean. Additionally, tiles make a sort of state machine by allowing tiles to be "destroyed" by certain damage types, resulting in the tile being replaced by a different tile based on the tile's data and the damage type. This allows the player to interact with the environment to open new paths using different abilities in an extensible way.

There are four coordinate systems in the game - the worldspace coordinates, the on-screen worldspace coordinates, the cellspace coordinates, and the screenspace coordinates, so I created many functions to convert between these as necessary. The return values of these are either a `p5.Vector` or an array of numbers, depending on which is more convenient. (the array can be immediately unrolled into another function argument, while the vector needs to be stored in a variable then have its attributes accessed for the function call)

Collision

Each entity has a collider type property (rectangle, circle, or cone), which is used to find the correct collision function from a function table. The collision functions used are fairly standard functions for checking whether orthogonal rectangles and circles overlap, with cone being a special case of circle with an angle check.

The Player Avatar

Control of the player avatar is straightforward. Each frame that one of the direction keys is held, the player tries to move at its speed in the combination of those directions. To determine where the player avatar is facing, a similar thing is done with the mouse coordinates, which are converted into world-space coordinates so a direction vector can be calculated between the player position and the mouse position.

Lighting (and the lamp)

As stated earlier, the `onScreenWorld` is used to calculate the lighting. Each cell in the world has a `light` attribute, which is used to determine the "damage" an enemy in that cell should take, as well as how the lighting should be drawn. Lighting is drawn using a separate WebGL graphics, and combined with the drawing graphics using the `blend` function to create the lighting effect. This also allows for the smooth lighting, as WebGL can interpolate the fill colour of shapes from the vertex colour, as well as having the enemies be unaffected by the lighting, so the glowing damage effect of the enemies can be created. This is done using `tint`, which would be very slow to apply to the whole screen, but is fine for the small number of enemy sprites on screen.

Because of this design, anything can contribute to the light level and affect the lighting, which creates the potential of other light sources in the world other than the lamp. I make use of this to create a dim glow around the player, as unlit areas must be rendered black to prevent the player from seeing through walls and this would make the player blind. This is done with a

simple distance calculation from on-screen cells to the player. For the lamp light, I do a similar thing. First, an angle check is done comparing the light's spread with the angle between the player facing direction and the direction to the cell, followed by a distance calculation using the light's power and falloff with the square distance to the cell. If the light level of a cell after these two calculations is non-zero, it then gets the list of cells on the path to that cell to test if there are any opaque cells in the way, in which case the light level is 0. This check is done last as it is the most expensive, and is pointless to calculate if the light level is already 0.

The `light` attribute defines the light level at the centre of each cell, so the vertex light level is calculated from an average of the four cells joining to make the vertex. This has the added effect of causing the light to bleed into opaque cells, creating the illusion that light is being cast onto the wall even though the wall is just a flat image.

Items

The items and behaviour are controlled through data and function tables, as with most things. There are essentially 5 types of item effects - lamp, attack, oil, teleport, none. The lamp is as explained above, with a limited oil supply. The oil refils the oil supply of a held lamp. The "attack" can also have multiple types, with properties such as "damage type" (defining how it affects tiles) defined in data. These attacks are the only dynamic entity, and are deleted based on a timer. A special collider type had to be created for the knife attack, as it has a cone-like shape, which is a special case of the circle collider that adds an angle check.

Enemies

Enemies navigate using the `updatePos` function, and accelerating toward to each entry in a list of cells. The acceleration is to make the enemy movement appear more smooth, rather than them instantly snapping to a new direction upon reaching a cell. This approach means a list of cells returned from A* search and a list of patrol cells defined in data can be treated equally. Enemies do not collide with anything, as they do not collect items and I chose to have them damage the player gradually by touching them as this gives a creepier feel. This means that two enemies sharing the path may end up overlapping, as they don't push away from each other, but this can be avoided by desining the enemy behaviour well, such as varying the enemy speeds and acceleration.

Enemy behaviour is defined through a FSM in their data. Each state has a list of transition functions and a corresponding list of states to transition to, as well as attributes for any required parameters for the current state. The current state of an enemy is used to get a function from the behaviour table, so enemies in the same state will act similarly so long as they continue to be in the same state, but these behaviour functions are often parameterised. For example, a common behaviour function is `patrol`, but the patrol path is entirely defined by the enemy's data.

Common transition functions include hearing, which is implemented as a square distance check, sight, which functions similarly to how lighting is blocked by walls, and whether a timer has reached zero. This approach makes it very easy to make diverse enemy behaviour, and the transition and behaviour functions could easily be extended by adding new functions to the tables, with no other code changes.

Game state

To switch between the main menu, game, inventory, and end screens, a game state variable is used, which is the key for function tables defining which update, draw and overlay functions to use. There are also functions called on entering and exiting each state, which control the sound and HTML elements.

UI

The interactive UI elements - the inventory, buttons and difficulty selector - are all done through HTML elements. p5 has a lot of functions to help with this, so it makes sense to use these existing interactible elements instead of creating my own. There are a couple of downsides to this though - firstly, these HTML elements can't use the images already loaded by p5, so the images for items have to be loaded a second time for the HTML elements, but are cached after that and secondly, there are some useful things that simply can't be directly styled, such as the dropdown menu and the hover text. There are function to create the appropriate HTML elements on entering a state, which are then stored in a container element in a variable so they can all be easily deleted on exiting the state.

Level Design

Plan for Level Design

We expect the player to backtrack to an area with a route they previously couldn't traverse as a core part of gameplay. To make this not frustrating, there are a few things that must happen:

1. the distance to backtrack shouldn't be too long - this can be aided by allowing the player to open shortcuts, either as they explore or using a new item, meaning they can go backwards through areas quickly, but there can still be a large area to explore.
2. the use of the item/ability to traverse such a route should be recogniseable, but not necessarily obvious. Making it not obvious allows the player to figure things out for themselves, increasing player satisfaction, but it needs to be reasonable to figure out by themselves. For instance, you could hint at a use of an item/ability by having an area next to where they collect it which requires or otherwise encourages them to try it out.
3. routes you can't traverse yet must be recogniseable as routes you may be able to traverse in the future, with new abilities - the extent to which depends whether this is a critical route,

or a hidden bonus area.

4. different areas of the environment should be easily distinguishable, and memorable, so the player can easily recall how to get to where they're going.

Secret areas can give the player secret items which unlock bonus routes, or provide a different method of traversing some routes (which may prevent them from needing to collect the usually used item). These bonus routes could lead to more secret items, or provide an alternate route along the critical path. It's good for there to be multiple ways to get to a destination.

The actual design

There are 5 abilities to let you traverse, associated with 5 items:

- Knife - a starting item, can cut through some things
- Shadow Dagger - an item hidden on a side path, cuts through illusory walls
- Ice Tome - the first easily accessible item, creates an ice projectile
- Fire Tome - creates a fire projectile
- Teleport Tome - allows you to teleport to any place you can see and fit in (hidden behind an illusory wall)

There are 7 types of tile obstacles:

- Thin bushes - can be cut through with several knife slashes, or instantly burned with fire
- Thick bushes - these can be cut with the knife after being frozen with the ice tome
- Brambles - can be thinned with the knife, allowing you to see and therefore teleport * through it, or instantly burned with fire
- Lava - this can be frozen with the ice tome to create a safe path
- Pits / Narrow tunnels - this can only be teleported past
- Illusory Walls - look just like regular walls, and can only be cut through with the shadow dagger, but the lighting allows to to just barely see that there's empty space behind them

Some of these tiles have multiple ways of getting past them, such as the lava and brambles, and the different abilities allow you to traverse different tiles. The Ice and Fire tomes also act as projectile attacks that briefly stun enemies, and the Teleport tome can help you get past enemies also, so these items give you abilities beyond simply acting like "keys" for getting past certain types of obstacle.

Notable Level Design Details

The starting area acts as a sort of tutorial, with the starting room immediately hinting at the exploratory nature of the game, with two exits and an oil pickup hidden locked behind some thick bushes. The closest enemy is fairly harmless, and guards two items, introducing the player to the idea of having to navigate past enemies to progress, and allowing them to experiment with ways of dealing with them. Additionally, from the starting area without clearing any obstacles, you can find 6 of the 7 types of tile obstacle (everything but the pit), introducing the player to the different obstacles they'll have to find items to overcome.

The intended route on a first playthrough is to find the ice tome (which is practically out in the open), which can be used to get past the lava lake to get on the path to the fire tome. This lava lake is large and bright in order to attract the player's interest. Following this path leads to a maze-like area with many enemies hiding, ending in the fire tome, placed right next to the brambles it can burn through to teach the player how to use it. The player can now travel back to the brambles they found from the starting area to head into the final area, which can be navigated with no other pickups.

There is an alternate minimal route, which is secret, which involves taking a side path to find the shadow dagger, and using that to cut through illusory walls and find the teleport tome. These two items are also all you need to go through the final area and reach the exit, creating two alternate minimal routes, which improves repayability.

If the player picks up all four of these items, they will be able to traverse a "secret" area near the final area, which gives them access to the "super lamp", the most powerful type of lamp that they can use to get through the final area more easily, and their ultimate reward for exploring. This isn't the only reward for exploring though, as there are many oil pickups in secret areas and side paths around the map, as well as two other lamps to find.

Evaluation and Debugging

There were two main stages of evaluation during development. The first stage had a simple test map, along with the player and a single enemy, so I had a simple environment to make the core functionality (i.e. wall collision, lighting, enemy behaviour, player movement, the camera), and could simply run the game to immediately test if these worked. In the second stage, I built the full game map, populated with the items and enemies, as the main thing to test was that the data for the items, world and enemies were correct. Since everything was easily controlled by the data, changing things such as the player starting position and starting items to immediately test different areas of the game and different items was easy. In both stages, I mainly focused on testing the functionality of the game during normal play, but I did test some more common edge cases. For example, having enemies stunned by both the light and an attack at the same time.

When debugging the problems I found, I often started by logging variables to the console to get more information, but for many problems this was not sufficient, as the amount of data logged

could be very large. I had two additional techniques I would use when this didn't work. The first is to print that same data to the screen instead, so I would constantly see the current state of a variable and observe how and when it changed. The second are a set of functions for helper graphics, rendering things such as the bounding box and centre position of entities to help with collision and whether entities are being rendered correctly, and drawing a dot in each cell that has a non-empty entities list to check if that structure was being updated correctly. These approaches were usually far more helpful than simply logging data.

Problems Encountered

- Collision etc. does not function properly when numbers are exactly on grid lines
  Add a very small offset to every position, making it very unlikely for the position to be exactly on a grid line
- Jittery y-scroll
  cause: cells were being offset based on the camera position, not the screen edges
  solution: fix the maths
- Incorrect wall collision (clip into wall, then have movement restricted by wall)
  cause: cells were being found based on the movement across the line, not the edge of the bounding box
  solution: add an edges array, and index into it
  cause: entity was being removed from cells based on the position after movement, not the current position
  solution: make a function which offsets the cells in the opposite direction of movement along the axis
  cause: edgeCorners was flipping the sign of the movement
- Enemies orbit a target cell
  The enemy only removes the cell from their plan once they get close enough, so if their acceleration is too small / their speed too great they will literally orbit the cell. This is solved by increasing the removal radius, increasing the acceleration, or decreasing the speed, and I usually solved it by increasing the acceleration as this is the simplest solution. However, since the acceleration and speed are defined in data, this orbiting effect is always possible to happen.
- `rect` draws a circle
  cause: automatic corner curve
  solution: write my own rectangle function
- `tint` is super slow
  Use a tinting rectangle instead (i.e., rectangles with transparency)

- The tinting rectangles have slight gaps between them sometimes
  Draw them on a separate WebGL canvas and mix it on top with `blend` and the "hard light" mixing mode (as transparency is very slow in WebGL)
- Drawing the rectangles is still very slow
  WebGL is slow at doing basically anything other than drawing a continuous shape out of triangles, so use `beginshape` with `TRIANGLE_STRIP` to draw the rectangles (which also allows for vertex shading to get smooth lighting)
- new/old cells not being found properly because each move direction is checked independently
  add the move first and check the new position, then undo the move
- Can use oil repeatedly in inventory
  reload inventory on item use using gameTransition
- A* doesn't consider that the enemy might be larger than 1 cell
  temporarily move the entity to the test cell and use entityCells to make sure it can traverse all cells
  it will be in if in the centre of that cell
- If an enemy can't be exactly in the centre of the player's cell, it thinks there is no path
  instead of checking if the current cell is exactly the goal cell in A*, check if and of the cells the entity occupies are the goal cell
- Game-breaking bug where transition functions that shouldn't be there appear in the enemy fsm
  using a variable to store the current fsm, because it's a for each loop so they state may change, so it accesses the `tos` list from a different state
- Attacks damage tiles multiple times
  add a hit timer to tiles that expires 1 tick after the attack
- Some enemies and items in the wrong position
  Likely human error, as these were manually positioned
  Generate the enemy starting position from the first entry in the `patrol` list instead of calculating it manually (this also defines where an enemy will retreat to if it has no patrol route, so it makes sense for it to be the starting position)
- Fire projectile doesn't move
  check the data - turns out mergeAttack only checks attack data and not item data for a speed value
- Projectiles don't hit walls properly
  make them all circles, as rectangle collision only works for orthogonal rectangles (still not

fixed)
add a traverse type to these damageable walls that allow the damaging projectile to go through them

- p5's image distortion functions are very slow
  Add a separate WebGL canvas which uses a shader to add the distortion effect, and modify the `render` function to switch between rendering this canvas or the main canvas depending on the game state
- Big performance hit if enemy can't find a path to the player, as the enemy tries to find a path every frame
  Only use the planTimer to replan the path, rather than recalculating on an empty plan
- Enemies struggle with corners and get stuck in walls sometimes
  cause: A* only considers that the entity is capable of existing in the centre of each cell on the path, and the optimal route often means getting as close to the wall as possible, which can cause collision issues
- Sound doesn't start until the user gestures on the page
  cause: This is a policy enforced by the web browser, there is no workaround.
  soution: using `userStartAudio()` means that the looping title music will start as soon as the user gestures
- large frame drop in the "maze"
  Unclear what's causing this. It's most likely one or more of the enemies, but I tried removing some of the nearby enemies to see which one might be the cause and didn't observe a consistent improvement. The most likely culprits seem to be the patrolling enemies in the maze, but it's unclear why they would cause this as they don't use any of the intensive functions.
- The framerate is cut in half if the computer is displaying on multiple screens
  The framerate is still stable, so I speculate this is just strange behaviour of p5 or the canvases rather than a problem in my code.

Playtesting

Aside from my own evaluation, I had two people play the game when it was close to finished, observed how they played the game and noted their feedback

- Feeback: the walls and floor are difficult to distinguish when not adjacent to each other
  I brightened the walls and darkened the floor to make them much more distinct
- Feedback: the corners are frustrating to navigate
  This is a consequence of the fundamental way the game is built (using tiles), so a proper

solution to this would be rather difficult. It is more of an inconvenience than a serious problem, however

- Observation: one player used items from the inventory frequently, which is perfectly functional but I didn't expect it, so item cooldown isn't shown in the inventory
  use CSS to dim inventory items when on cooldown (no gradual tint like with equipped items, as the timer doesn't run while the inventory is open so this wouldn't be very helpful)
- Observation: neither player figured out what the "shadow dagger" item did
  this isn't a huge issue, as it is supposed to be a secret, but I did add an item of a new type - a note, giving a hint at the illusory walls the dagger destroys
- Observation: neither player managed to beat the game
  They both played on the normal difficulty, and the level of difficulty observed is the intended challenge for that mode. My personal playtesting has shown that the game is beatable on the highest difficult using both main routes.
- Observation: neither player tried to use the light or attacks to slow down the enemies, opting to instead rush through an hope for the best
  I modified the enemy graphics so that the light damage was more visible, and made it so light was more effective on the normal and easy difficulty, so the player would learn this mechanic more quickly.

Possible Extensions

Save files and level-streaming

Since the status of the game is entirely held within JSON-like objects, the status of the game could be quite easily saved by saving these objects as JSON files. This would result in storing a lot of redundant data, but the total size of these files won't be more than a few Mb, so I don't think that's a huge problem. For loading the save files, a nice solution would be to have the main menu be a separate browser page from the game, so you can make use of the async guarantees that `preload` gives for loading files. Passing the information for which files should be loaded between the pages shouldn't be difficult, as there are standard ways for doing this.

The current size of the game isn't big enough to worry about using a level-streaming approach to prevent a long initial load time, but if the game were to get significantly bigger this may be worth considering. The data representing the world, items, enemies etc. are just json files, which won't be larger than a few Mb even for a very large world, so the actual level data can all be loaded at once - the concern is graphics and sound. These could be fetched on demand when the entities or tiles requiring them are close by (and possibly unloaded when far enough away, though the amount of storage sprites and sound files take up is likely not significant enough for

this to be a concern). The functions for loading files in p5 are already asynchronous, so the only concern is tracking whether the data is loaded (which can be handled with a callback function), and handling what should happen if the data isn't loaded by the time it's needed. The standard approach for this is to have placeholders for missing graphics that are always loaded, and to simply play no sound when a sound is missing (as the sound doesn't usually give much information to the player).

Parallax environment

Instead of using fixed tile graphics for the environment, have wall, floor, and ceiling elements. The wall is tied to points in the ceiling and floor, which are used to distort the graphics to match the perspective, while the ceiling is effectively just scaled up proportional to its distance from the ground. There is no built-in skew function in p5, so this will either have to not use images, or I will have to write my own skew code. This is also a completely different way of doing level graphics, so the method of drawing the environment should be constructed in a way that this could easily replace it, same with the way the environment graphics are referenced in the Json/leveldata object. Could definitely make an interesting extension.

Stealth Mechanics

Currently, "hearing" is just a distance check, but this could be extended to account for walls using some sort of pathfinding, and to have stealth-like mechanics by attaching a noise level to different player actions. The player would have a "noise" attribute, which would have a base level and be modified by actions taken by the player, such as using certain items. This would act as a multiplier to enemies' hearing, allowing them to effectively hear the player from further away depending on the player's actions.

Hallucinations

Currently, the only affects of the madness buildup are the ripple effect of the screen, and a static sound. There are other madness effects (which I call hallucinations) that could be added, however. A lot of these effects would involve the creation of duplicate data structures - one representing reality, one representing what the player sees. For instance, if a hallucination causes light to show differently, an additional light value for each cell would need to be added, or at least a calculation changing how the light is rendered.

Enemy Hallucinations

One way to implement this is as a special type of enemy, which is only visible when the player exceeds a certain madness level, and is incapable of harming the player.

The effect could be enhanced by allowing such enemies to do illusory damage to the player - i.e. they appear to have hurt you, but the damage disappears when your madness level lowers

enough

Another enhancement would be to have these enemies appear non-deterministically - the madness level only defines a *chance* that the enemy will appear, and they may disappear again afterwards. This appearing/disappearing could be done both off-screen, and when the enemy is in darkness (which can be found by the light level).

Meta Hallucinations

Eternal Darkness had some madness effects that looked like your TV was acting up, such as appearing to change the volume or the source. This isn't as easy to do in a browser window, but there are still some effects like this you could do. For instance, you could cause the cursor symbol to change or disappear, or make the page look like its reloading.

Sound Hallucinations

This could work well with the additional of dynamic sound, for instance, and could allow changes that would normally be based on enemy proximity, for example, to also randomly happen based on your madness level, making it seem like there's an enemy nearby you can't see. If sound effects not triggered by the player (such as for enemy attacks) were added, these could also be used like this.

HUD/Inventory Hallucinations

There are many things that could be done here. The icons for the items could be changed to be slightly off, item descriptions could be changed, lamps could show the incorrect amount of fuel or it could show incorrect equipped items

Dynamic Music

This is a common feature of modern games - having music that changes based on what's currently happening. A simple way to implement this would be to separate out the background music into different "stems", and to set the volume of each stem based on the state of the game - for instance, how nearby an enemy is, the player's health or madness level, whether the lamp is on. Given how this is handled by p5, however, it's possible these stems will get out of sync as they loop, which will ruin the music.

Enemy Memory

Enemies can already have psuedo-memory through the state machine, but more explicit enemy could be added to create smarter or more interesting enemies. One noteable example is remembering a location

Obstacles

This is an additional type of entity I didn't end up implementing. This is essentially a catch-all for anything that isn't an enemy, tile, attack or item. This is how something such as a moving platform or interactible switches would be implemented, to add more environment interactions, but I decided the tile and item interactions were sufficient for the scale of this project, so it wouldn't be worth the significant extra work to create a new entity type like this.

Conclusion

One thing I definitely wished I could improve is the code organisation. Some files are rather large and messy (such as the "collision" file), and where certain multipurpose functions should be found is unclear. Additionally, while I tried to JSDoc every function, both to make development easier and to add to the readability of the code, I ended up not having time to fully JSDoc and clean up the functions added later in the project.

The biggest improvement I would want to make on this project is to expand the game and further refine the level design and gameplay. In terms of comparing this project to an actual game, I would consider the submission to be more like a proof of concept of the engine and game idea than a full game (though it does meet the criteria of being a complete game - there are opponents, failure states, and a goal). Thanks to the data-oriented design, a lot of content could be added to the game without writing any extra code, though I would want to build a simple program to help with building the enemies and items and such, as working with raw JSON isn't the nicest development environment. With this, and all the gameplay parameters, an almost endless time could be spent refining the smallest details of gameplay and enemy behaviour.

Ultimately, I consider this project to be a successful execution of the plan. While there are a lot of planned mechanics that are not implemented, the fundamental concept of the game fully is, and the playtesting shows it to be successful.