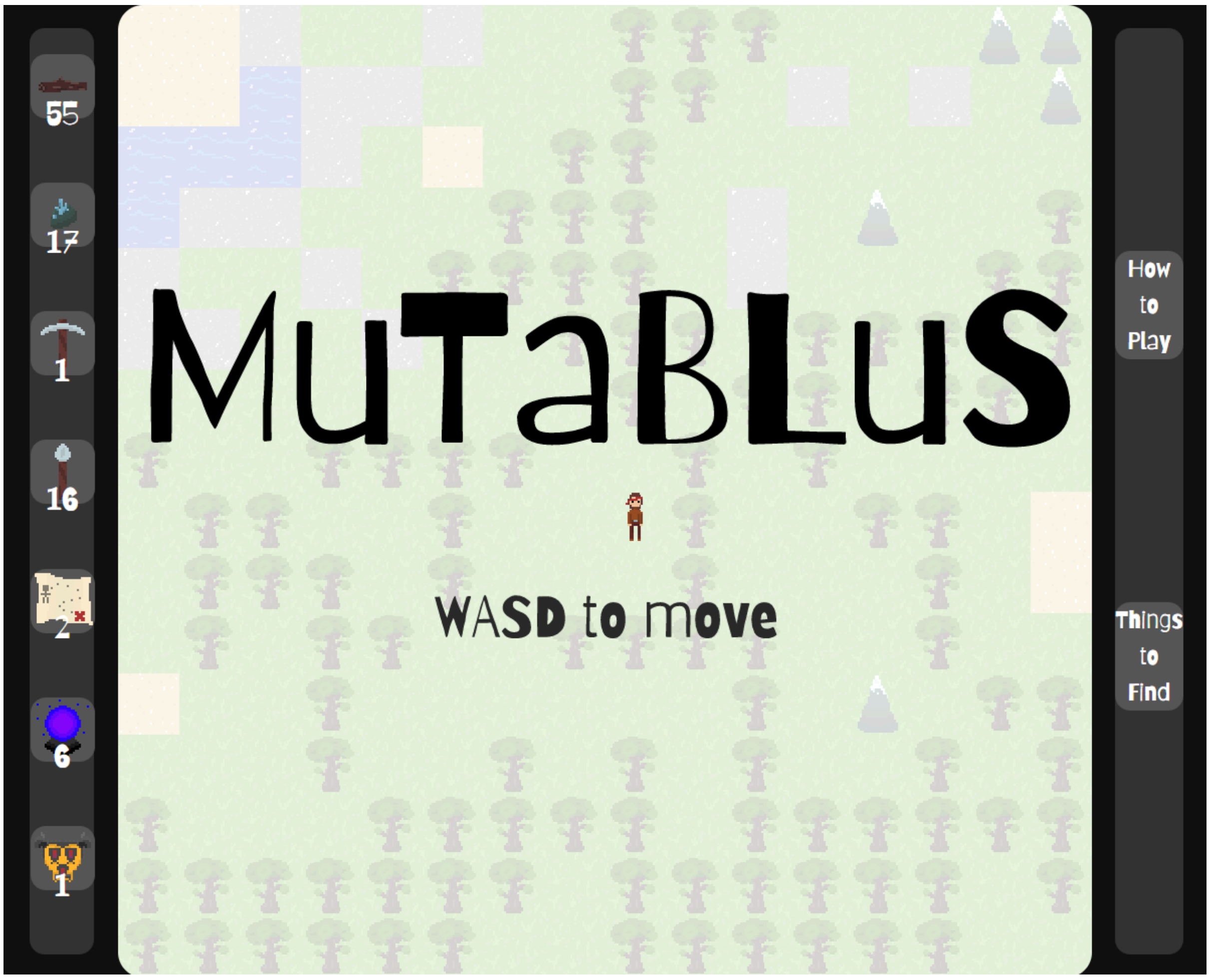
Mutablus

This project may be marked at 1280x800 or 1920x1080. The user experience will not change.

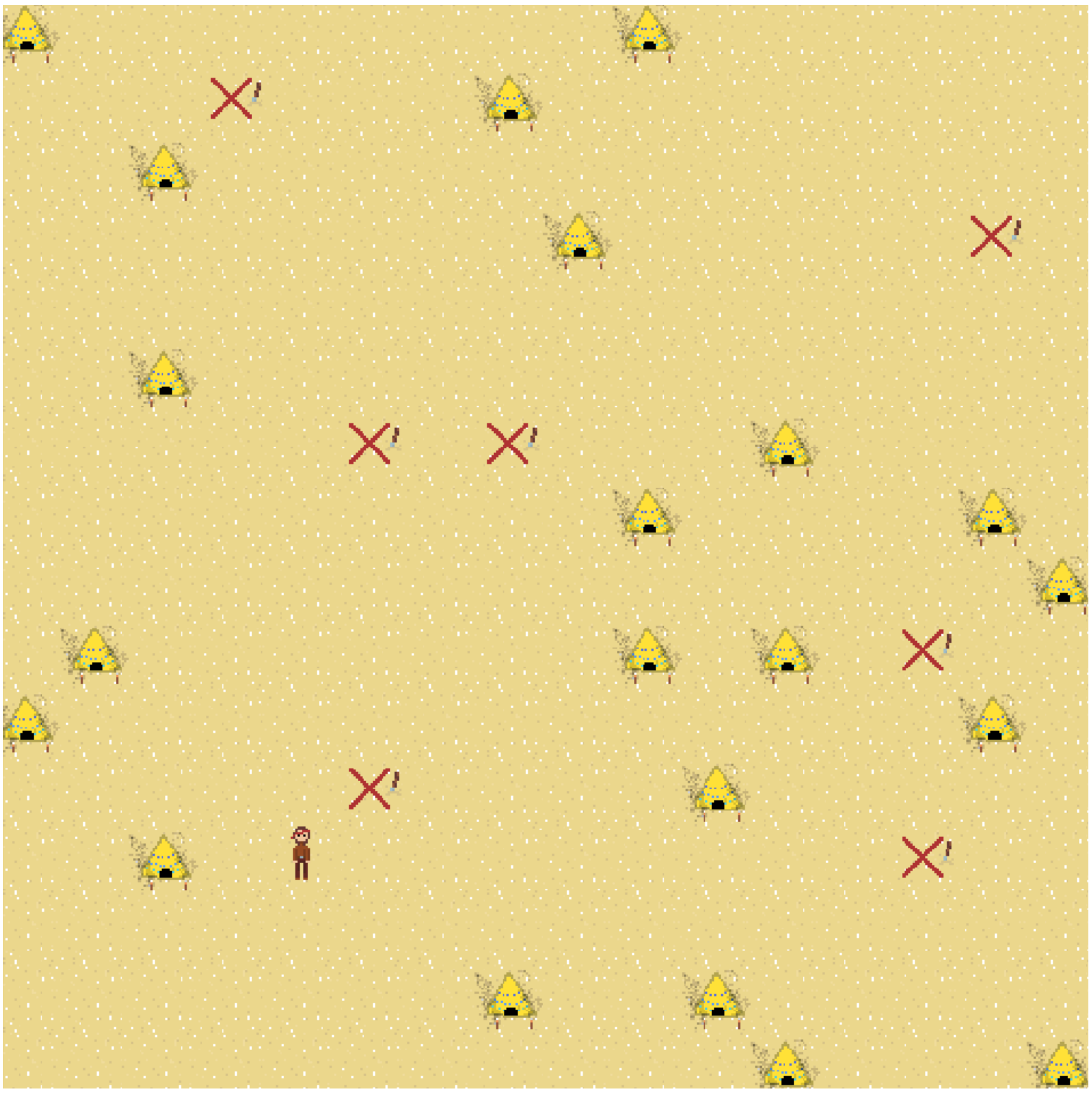
Overview



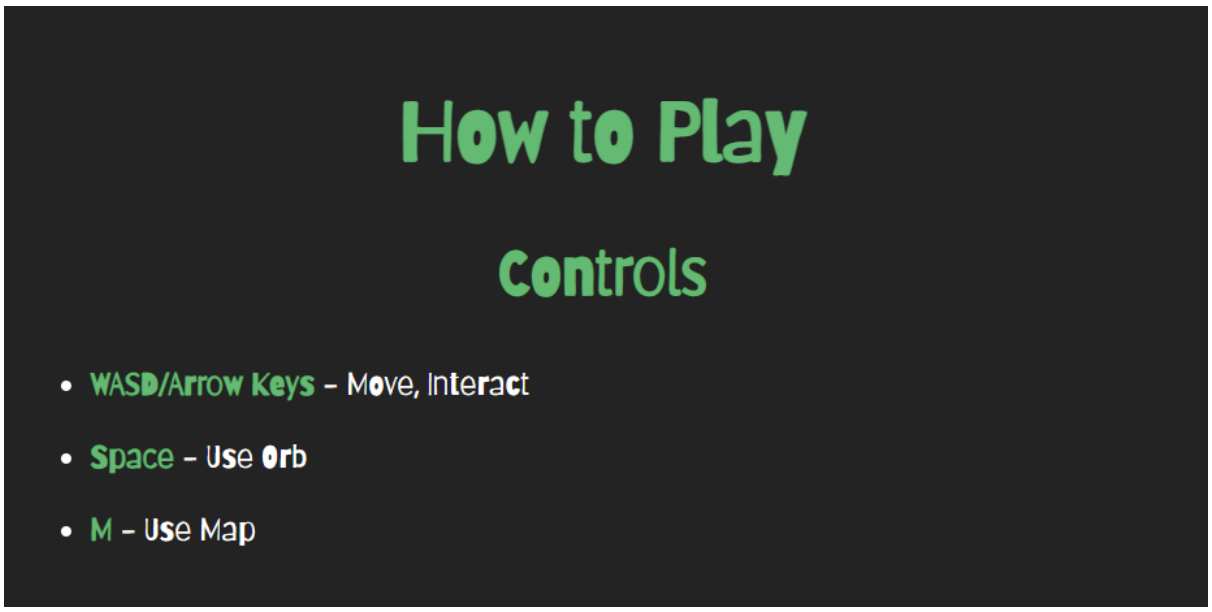
Title screen

Mutablus is a top-down exploration and item management game set in an ever-changing, procedurally generated world. The player controls a character who can move around the screen and interact with the world. The world contains different biomes, structures, and items for the player to find. Players can learn more about the game in the “How to Play” and “Things to Find” tabs. The game is inspired by old-school exploration games like *Adventure* and the *Ultima* series as well as the wave function collapse (WFC) algorithm and its use in Oskar Stålberg’s games like *Bad North* (Gumin, 2016). *Mutablus* aims to revitalise a lost genre by introducing newer game elements like item management and procedural world generation. This project’s technical side primarily deals with procedural world generation in the form of seed generation, WFC, and quality passes.

Usability and User Experience


Heavily populated Necropolis

As I aimed to develop a game with limited resources, I looked to older genres for inspiration. I settled on building an *Adventure*-style exploration game with elements from games of recent decades like *Minecraft* and *Dwarf Fortress*. The primary challenges of this type of game would be world generation and accessibility. I quickly decided procedural world generation would create the core of this game and developed a WFC algorithm based on Martin Donald's WFC rendition (Donald, 2020). I would have loved to create retro tile art influenced by games like *Ultima III: Exodus* and *Dwarf Fortress* but could not justify it with accessibility for wide demographics in mind. Instead, I opted for a more realistic, straightforward representation of tiles.

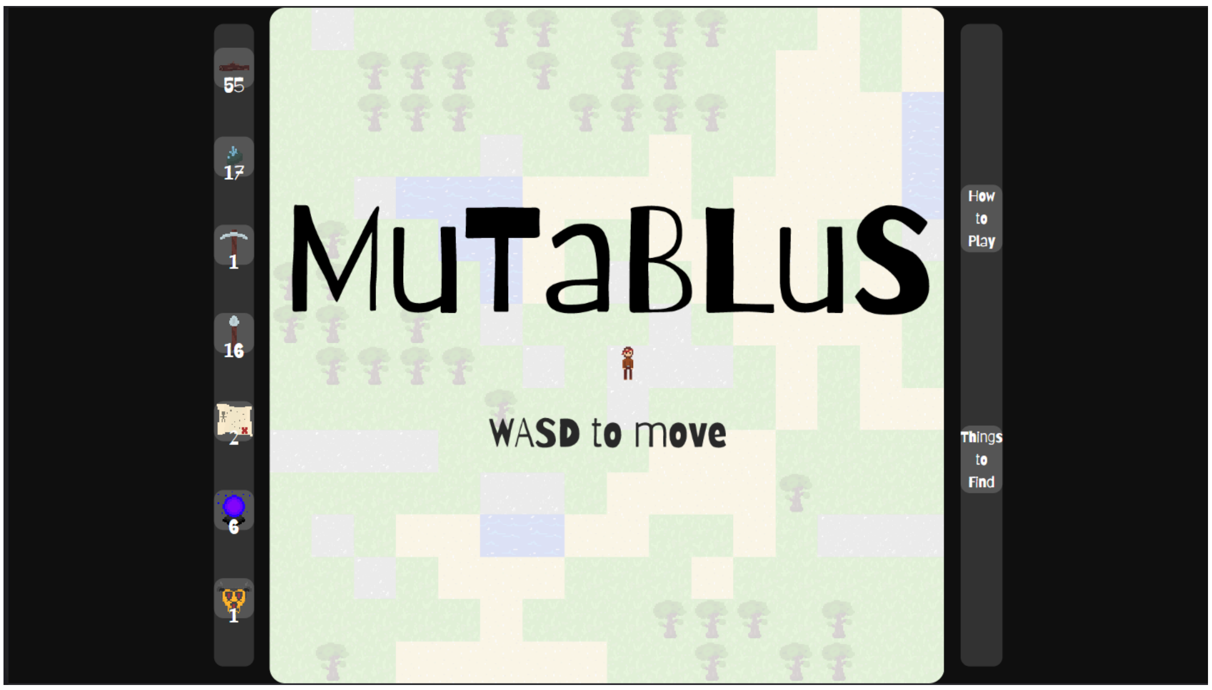

How to Play

Controls

- WASD/Arrow Keys – Move, Interact
- Space – Use Orb
- M – Use Map

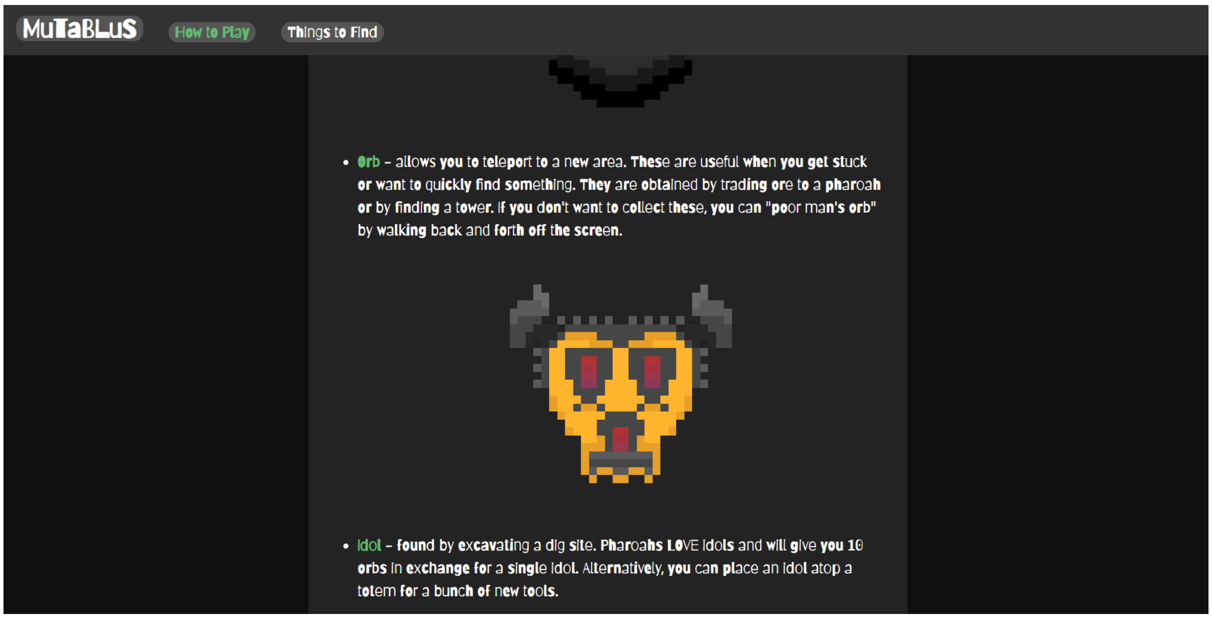
"How to Play" control scheme

Mutablus has simple modes of interaction, and thus, simple input. The player can move around the screen with WASD or arrow keys. Walking offscreen will take the player to a new area. I packed more functionality into these keys by making tiles interactable by walking into them. This prevented the need for a separate button and simplified the actions required from the player. Unfortunately, I did need to add two more inputs for item usage: spacebar for orbs and M for maps. By the nature of developing a 2D game, I did not need to go out of my way to make *Mutablus* entirely keyboard accessible. By only having two clickable buttons on the main screen (three on the information pages), the website is very accessible and easy to navigate with a keyboard.



Screen layout (1920x1080)

The game is limited to a small 16x16 grid, reducing load times and making the game visible on smaller screens. The game's square viewport is easy to fit in many resolutions and could be ported easily to mobile devices in portrait orientation (this is not currently implemented). I placed the inventory and information tabs outside the game viewport to give it as much of the screen as possible while fitting the navigation bars beside it in empty screen space.

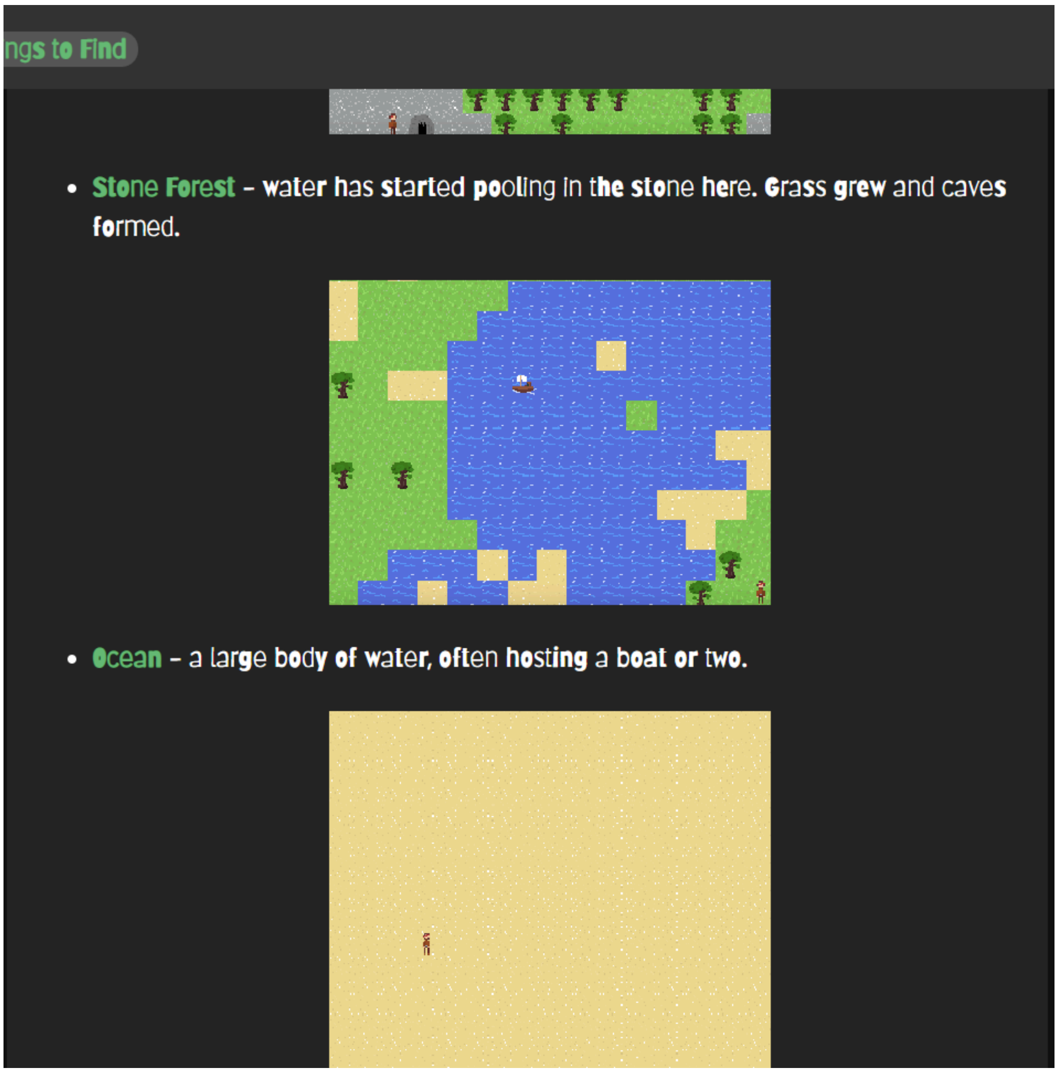


- Orb – allows you to teleport to a new area. These are useful when you get stuck or want to quickly find something. They are obtained by trading ore to a pharoah or by finding a tower. If you don't want to collect these, you can "poor man's orb" by walking back and forth off the screen.

- Idol – found by excavating a dig site. Pharoahs LOVE idols and will give you 10 orbs in exchange for a single idol. Alternatively, you can place an idol atop a totem for a bunch of new tools.

Item descriptions in "How to Play"

"How to Play" contains more basic information while "Things to Find" has more in-depth descriptions of rarer structures and biomes. These tabs are divided to allow players to read the tutorial and make non-essential discoveries themselves before reading about them in "Things to Find".

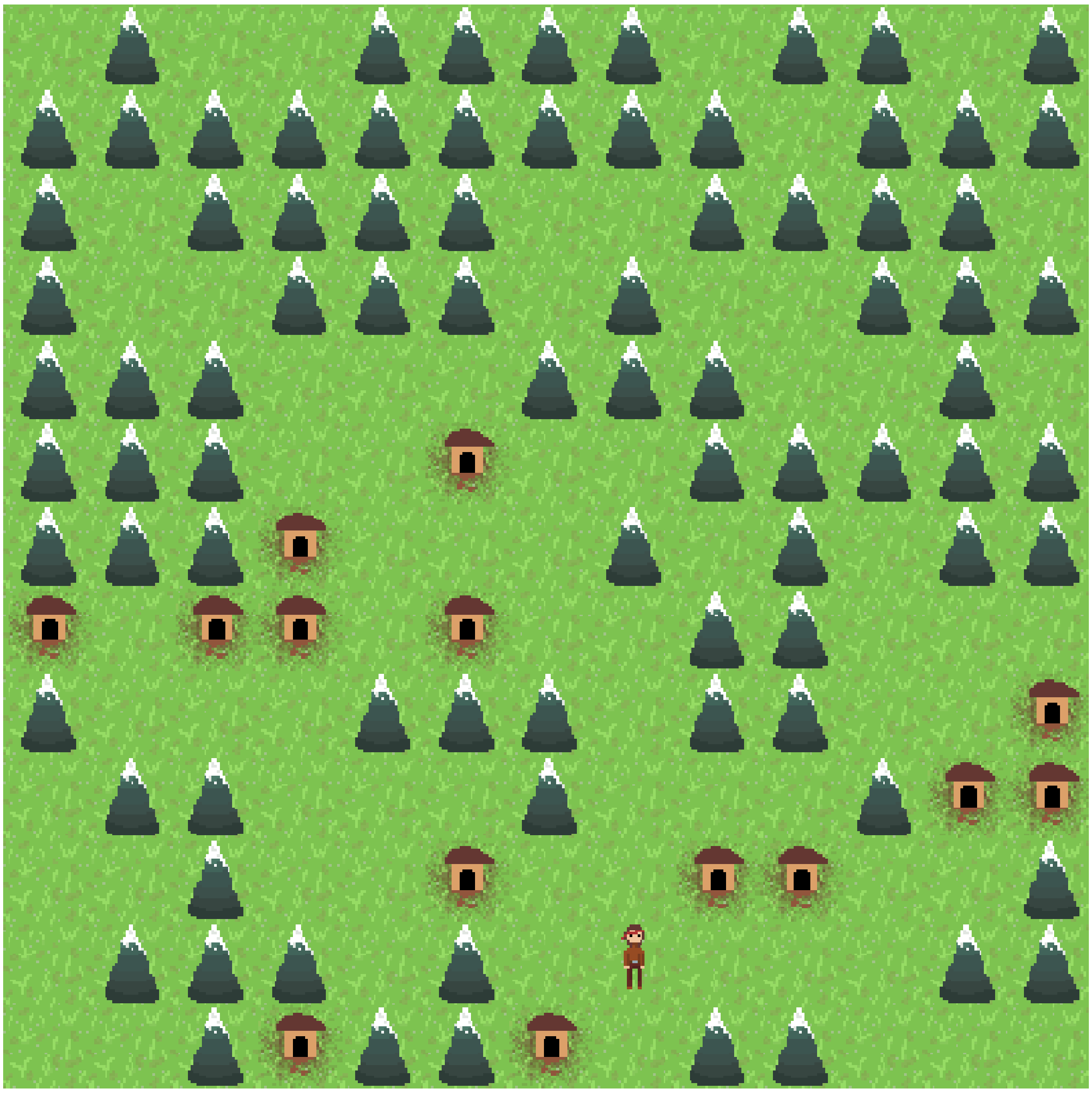

ngs to Find

- Stone Forest – water has started pooling in the stone here. Grass grew and caves formed.

- Ocean – a large body of water, often hosting a boat or two.

Screenshots help players know what to look for

Interesting seeds, structures, and points of interest are outlined in the “Things to Find” tab. With these objectives outlined in a separate but official manual, I aimed to maintain the culture of pre-Internet games by encouraging players not to search for solutions to problems online. Placing all the information the player needs a click away means they can explore the world for themselves and get help whenever they see fit. The slow feed of information through the vertical slice of content in the centre of the screen encourages players to stop scrolling when they have found what they are looking for. This way, they are less likely to spoil themselves on later game content. Rarer objectives are also placed nearer to the bottom of the page for this reason.

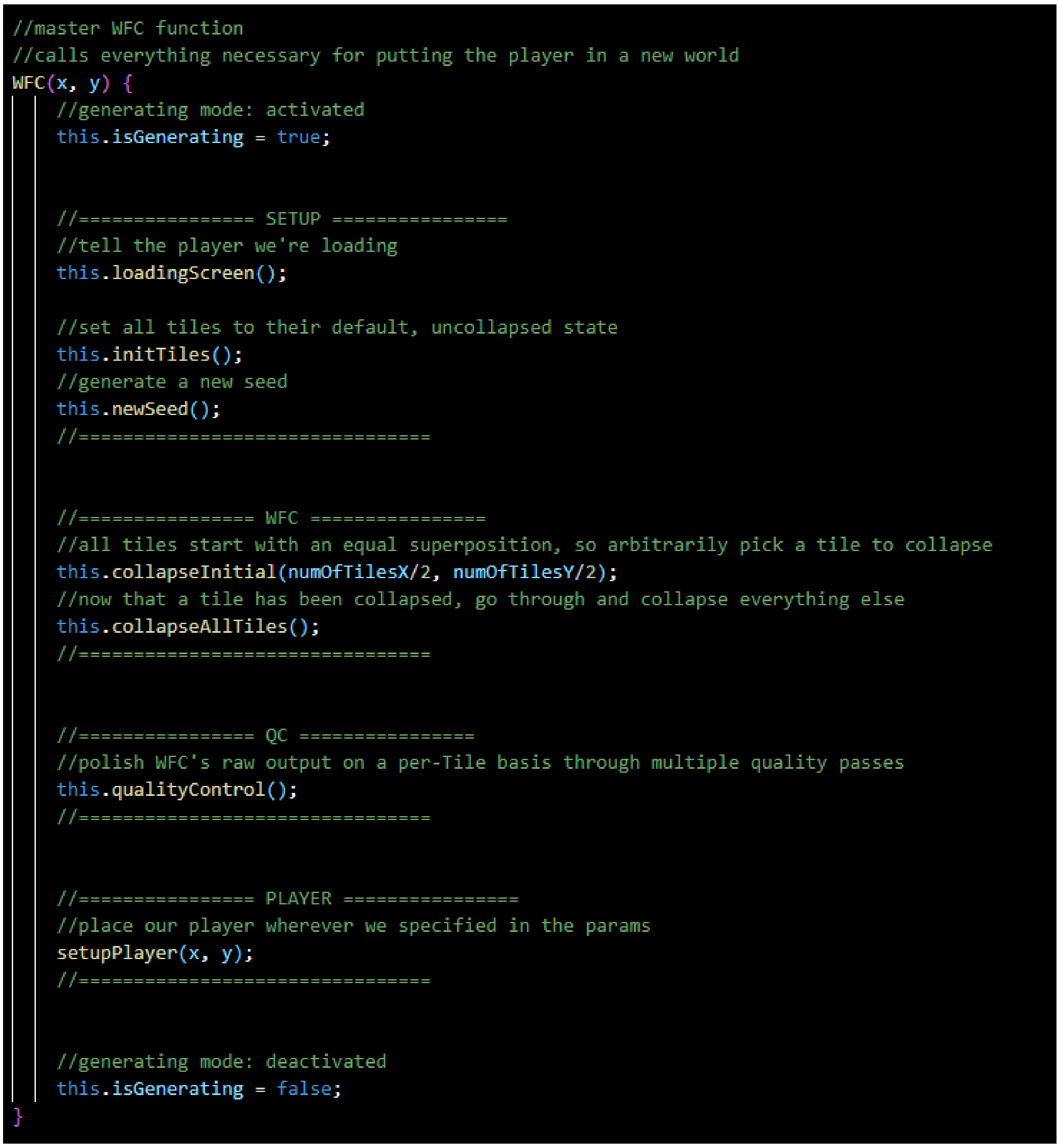


Mountain city

Some things are rare and take longer to find. For example, the mountain city biome has a 1/45 chance of spawning (without accounting for the chance that other criteria behind the scenes are met). Simpler structures like houses and boats can be tricky to find for players who get unlucky. While many players will not see the entirety of *Mutablus*' content, I wanted to create unique, memorable experiences for players who got lucky. The rarity of some points of interest should intrigue players, encouraging them to keep exploring.

Technical Implementation

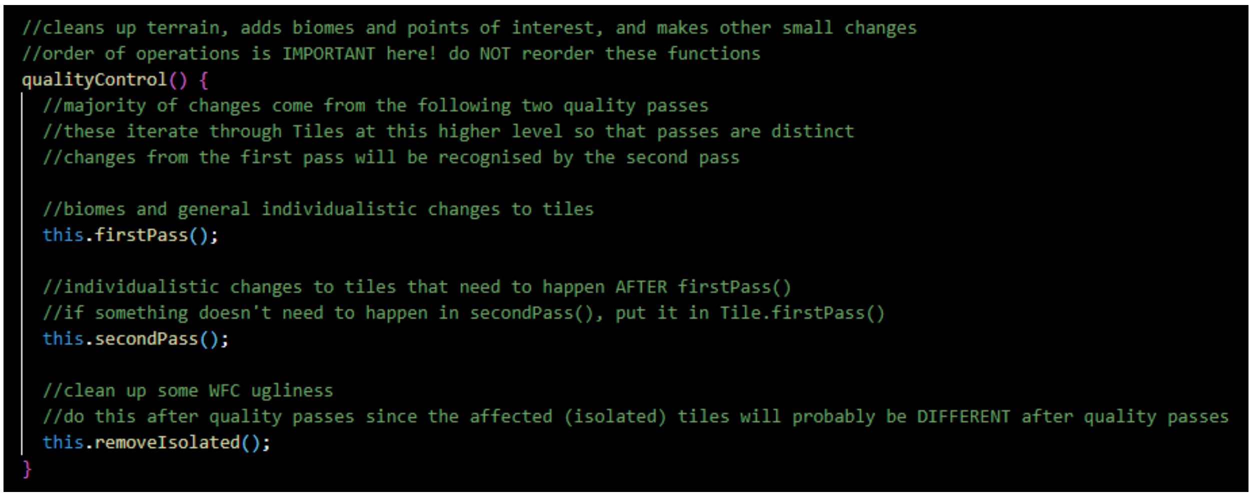
Mutablus’ code is mostly dedicated to world generation and tile interaction. The game uses my reverse-engineered WFC algorithm to generate a seed-based 2D world without Perlin noise. This game’s JavaScript uses elements of the *p5.js* plugin like setup(), draw(), and img() to draw the game on a canvas.



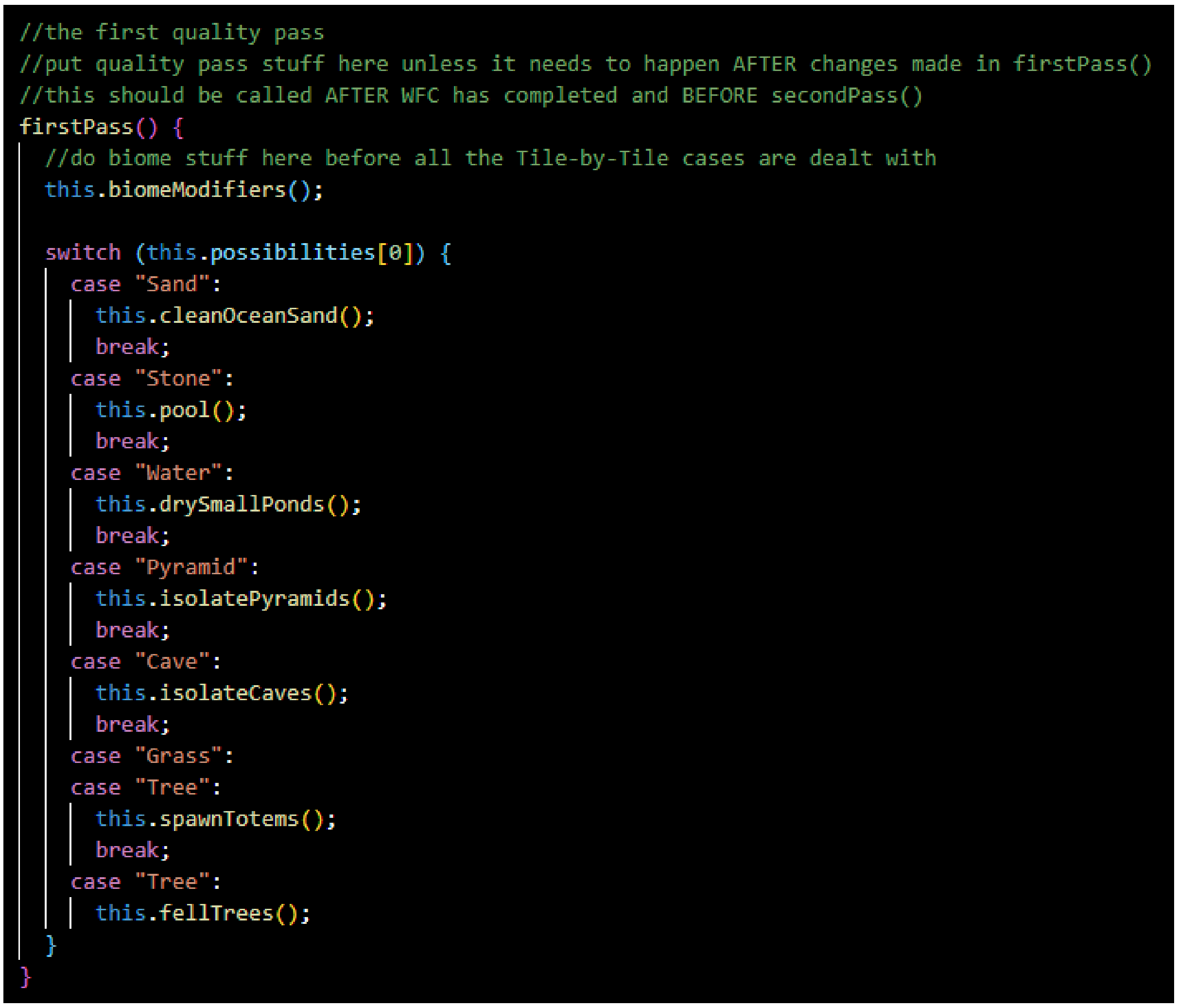
```
//master WFC function
//calls everything necessary for putting the player in a new world
WFC(x, y) {
    //generating mode: activated
    this.isGenerating = true;


    //================ SETUP ================
    //tell the player we're loading
    this.loadingScreen();

    //set all tiles to their default, uncollapsed state
    this.initTiles();
    //generate a new seed
    this.newSeed();
    //=================================


    //================ WFC ================
    //all tiles start with an equal superposition, so arbitrarily pick a tile to collapse
    this.collapseInitial(numOfTilesX/2, numOfTilesY/2);
    //now that a tile has been collapsed, go through and collapse everything else
    this.collapseAllTiles();
    //=================================


    //================ QC ================
    //polish WFC's raw output on a per-Tile basis through multiple quality passes
    this.qualityControl();
    //=================================


    //================ PLAYER ================
    //place our player wherever we specified in the params
    setupPlayer(x, y);
    //=================================


    //generating mode: deactivated
    this.isGenerating = false;
}
```

My high-level WFC implementation

For our purposes, WFC generates a grid of confirmed tile identities by “collapsing” a tile to one of several possibilities, propagating appropriate changes to neighbours, and repeating. Imagine there are three tile possibilities: grass, sand, and water. Grass can be adjacent to sand, sand can be adjacent to grass and water, water can be adjacent to sand, and any tile can be adjacent to itself. Each tile is initialised with an array of all three possible tile types. To begin, one tile is collapsed, meaning it randomly selects one of its possibilities and removes all other possibilities. For example, if this initial tile collapsed to the random possibility of “Grass”, the game would then recognise it as a confirmed grass tile. After collapsing, the tile then propagates its changes to adjacent tiles. Those tiles then propagate the changes to their possibilities to their neighbours and so on. This means that any tiles adjacent to our collapsed grass tile would lose the possibility of being water (as we have stated that grass cannot be adjacent to water). This leaves adjacent tiles with two possibilities: grass and sand. The algorithm then iterates through all tiles, identifies the tile with the least possibilities, collapses it, propagates changes from that collapse, and repeats until the whole grid consists of collapsed tiles. After this three-tile algorithm completes, the grid would consist of grass, sand, and water tiles with no grass tiles adjacent to water tiles. These steps paraphrase Watt Designs’ WFC explanation (Watt Designs, 2023).



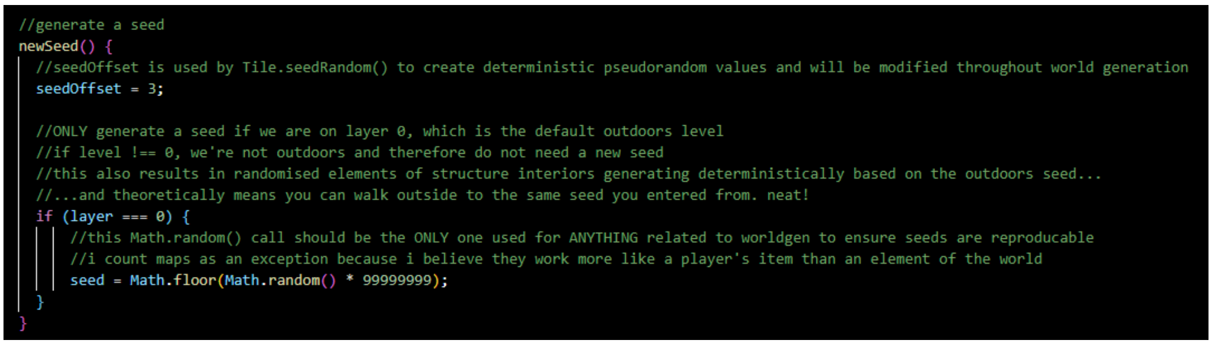
```
//cleans up terrain, adds biomes and points of interest, and makes other small changes
//order of operations is IMPORTANT here! do NOT reorder these functions
qualityControl() {
  //majority of changes come from the following two quality passes
  //these iterate through Tiles at this higher level so that passes are distinct
  //changes from the first pass will be recognised by the second pass

  //biomes and general individualistic changes to tiles
  this.firstPass();

  //individualistic changes to tiles that need to happen AFTER firstPass()
  //if something doesn't need to happen in secondPass(), put it in Tile.firstPass()
  this.secondPass();

  //clean up some WFC ugliness
  //do this after quality passes since the affected (isolated) tiles will probably be DIFFERENT after quality passes
  this.removeIsolated();
}
```



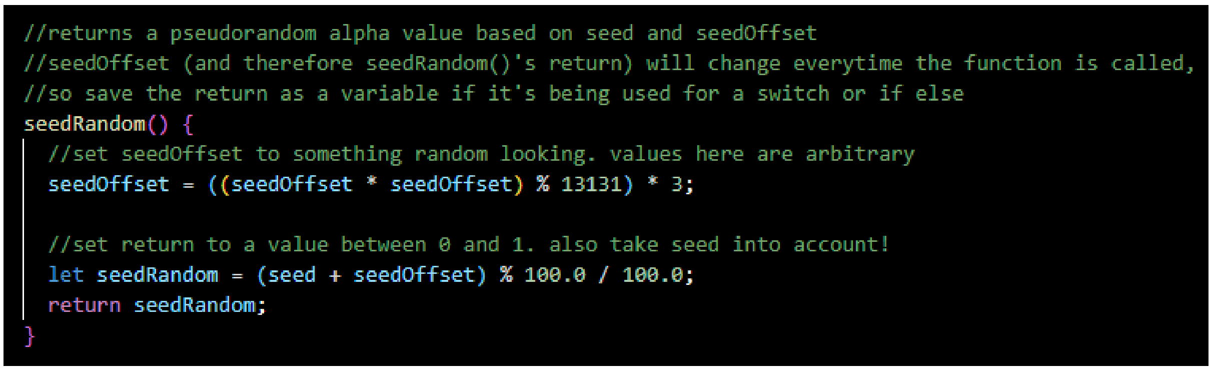
```
//the first quality pass
//put quality pass stuff here unless it needs to happen AFTER changes made in firstPass()
//this should be called AFTER WFC has completed and BEFORE secondPass()
firstPass() {
  //do biome stuff here before all the Tile-by-Tile cases are dealt with
  this.biomeModifiers();

  switch (this.possibilities[0]) {
    case "Sand":
      this.cleanOceanSand();
      break;
    case "Stone":
      this.pool();
      break;
    case "Water":
      this.drySmallPonds();
      break;
    case "Pyramid":
      this.isolatePyramids();
      break;
    case "Cave":
      this.isolateCaves();
      break;
    case "Grass":
    case "Tree":
      this.spawnTotems();
      break;
    case "Tree":
      this.fellTrees();
  }
}
```

Quality control

My algorithm works similarly to the generic one described above with some modifications: there are more Tile possibilities, changes are only propagated one Tile away from collapsed Tiles, randomness is seed-based, Tiles are collapsed using hard-coded neighbour and weighting systems, and the grid goes through a few quality control passes after all tiles are collapsed. Many of *Mutablus*' Tiles and biomes are not created during WFC and are added after the fact. Unique biomes like mountain ranges simply replace preexisting Tiles with other Tiles. Snowy areas, for example, replace grass with snow and water and boats with ice. Towers do not naturally generate, instead replacing preexisting houses and pyramids in frozen biomes. Moving non-essential work to post-WFC quality control passes reduces the number of possibilities WFC has to account for and speeds up loading times.



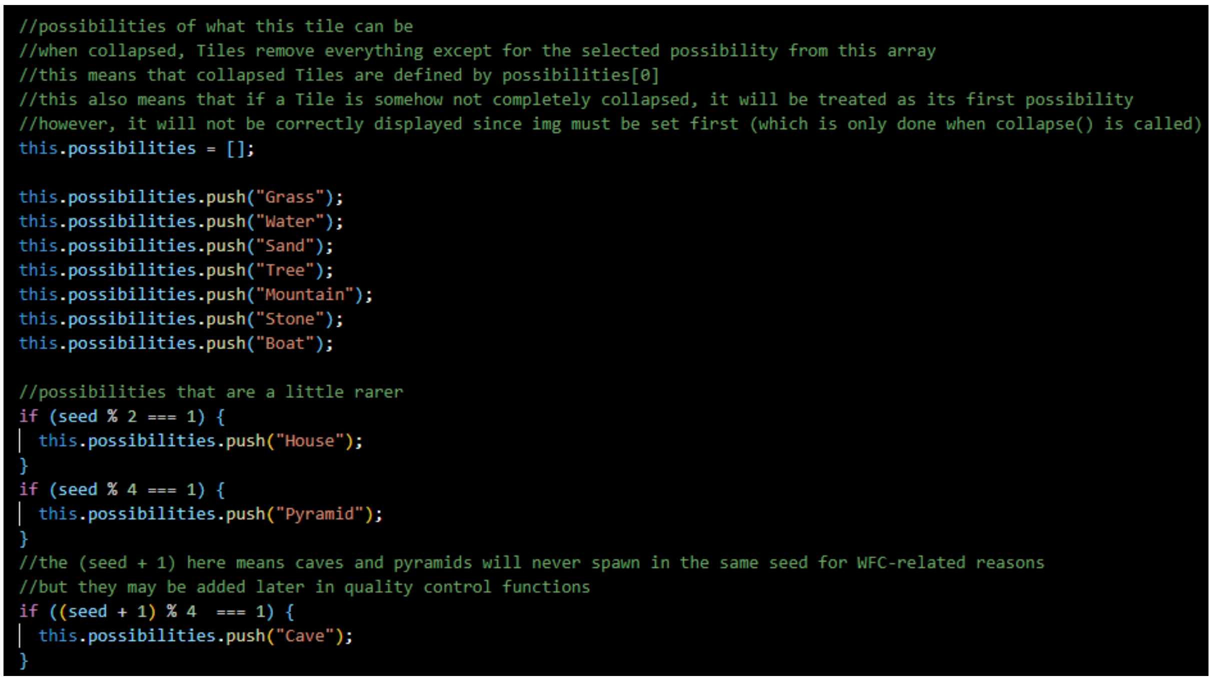
```
//generate a seed
newSeed() {
  //seedOffset is used by Tile.seedRandom() to create deterministic pseudorandom values and will be modified throughout world generation
  seedOffset = 3;

  //ONLY generate a seed if we are on layer 0, which is the default outdoors level
  //if level !== 0, we're not outdoors and therefore do not need a new seed
  //this also results in randomised elements of structure interiors generating deterministically based on the outdoors seed...
  //...and theoretically means you can walk outside to the same seed you entered from. neat!
  if (layer === 0) {
      //this Math.random() call should be the ONLY one used for ANYTHING related to worldgen to ensure seeds are reproducable
      //i count maps as an exception because i believe they work more like a player's item than an element of the world
      seed = Math.floor(Math.random() * 99999999);
  }
}
```



```
//returns a pseudorandom alpha value based on seed and seedOffset
//seedOffset (and therefore seedRandom()'s return) will change everytime the function is called,
//so save the return as a variable if it's being used for a switch or if else
seedRandom() {
  //set seedOffset to something random looking. values here are arbitrary
  seedOffset = ((seedOffset * seedOffset) % 13131) * 3;

  //set return to a value between 0 and 1. also take seed into account!
  let seedRandom = (seed + seedOffset) % 100.0 / 100.0;
  return seedRandom;
}
```

Seed generation and use in Tile.seedRandom()

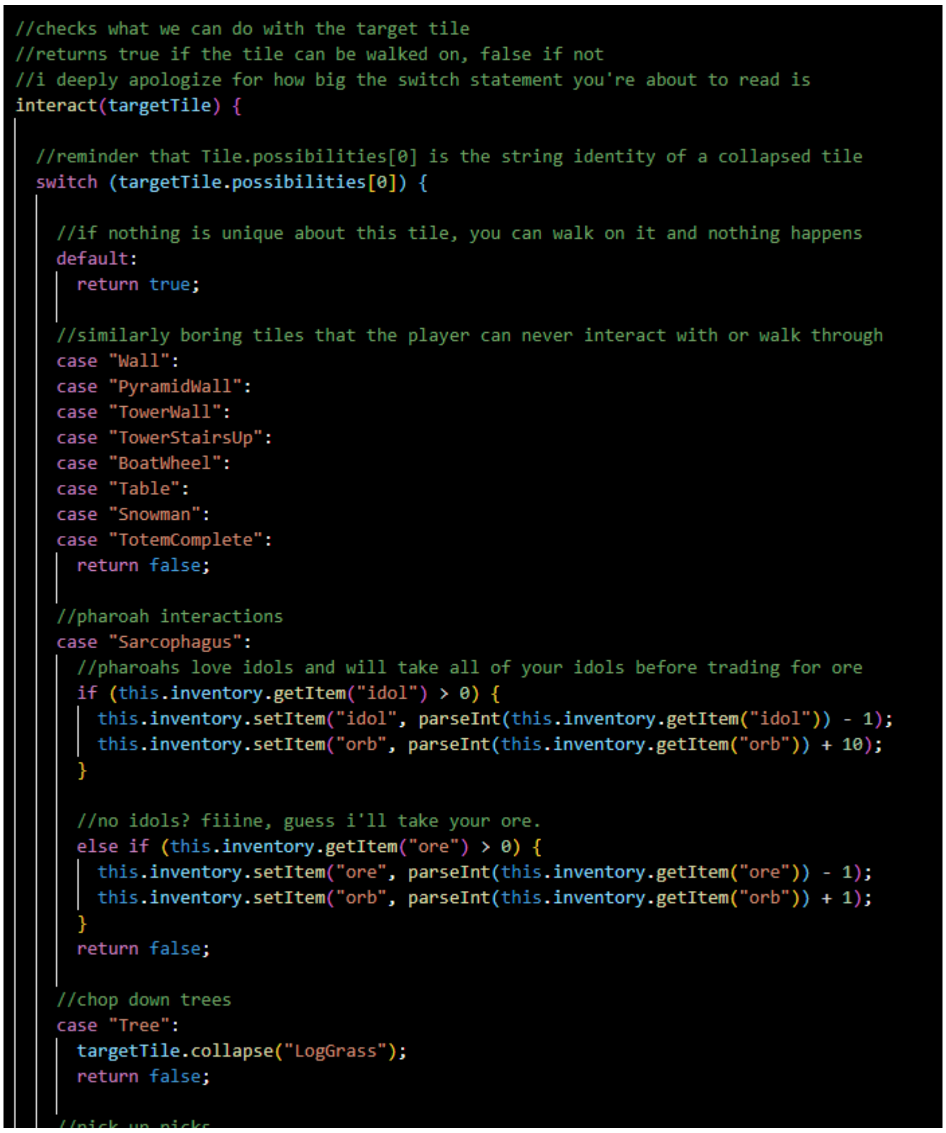
Rather than randomly collapsing Tiles and changing the world, a seed is randomly generated before the WFC algorithm runs. Tile.seedRandom() replaces the Math.random() function by providing a random value based on the world seed. seedRandom() is meant to return a deterministic, pseudo-random alpha value. This way, worlds and structure interiors can be recreated by using the seed while still providing unique outputs. There is an issue with my implementation that often causes seeds to generate non-deterministic worlds. Unfortunately, I could not solve this problem, so I built it into *Mutablus*' mutable world narrative by warning players they may not always exit structures to find the same place they entered from.



```
//possibilities of what this tile can be
//when collapsed, Tiles remove everything except for the selected possibility from this array
//this means that collapsed Tiles are defined by possibilities[0]
//this also means that if a Tile is somehow not completely collapsed, it will be treated as its first possibility
//however, it will not be correctly displayed since img must be set first (which is only done when collapse() is called)
this.possibilities = [];

this.possibilities.push("Grass");
this.possibilities.push("Water");
this.possibilities.push("Sand");
this.possibilities.push("Tree");
this.possibilities.push("Mountain");
this.possibilities.push("Stone");
this.possibilities.push("Boat");

//possibilities that are a little rarer
if (seed % 2 === 1) {
  this.possibilities.push("House");
}
if (seed % 4 === 1) {
  this.possibilities.push("Pyramid");
}
//the (seed + 1) here means caves and pyramids will never spawn in the same seed for WFC-related reasons
//but they may be added later in quality control functions
if ((seed + 1) % 4  === 1) {
  this.possibilities.push("Cave");
}
```

Certain possibilities only enter the pool in some seeds

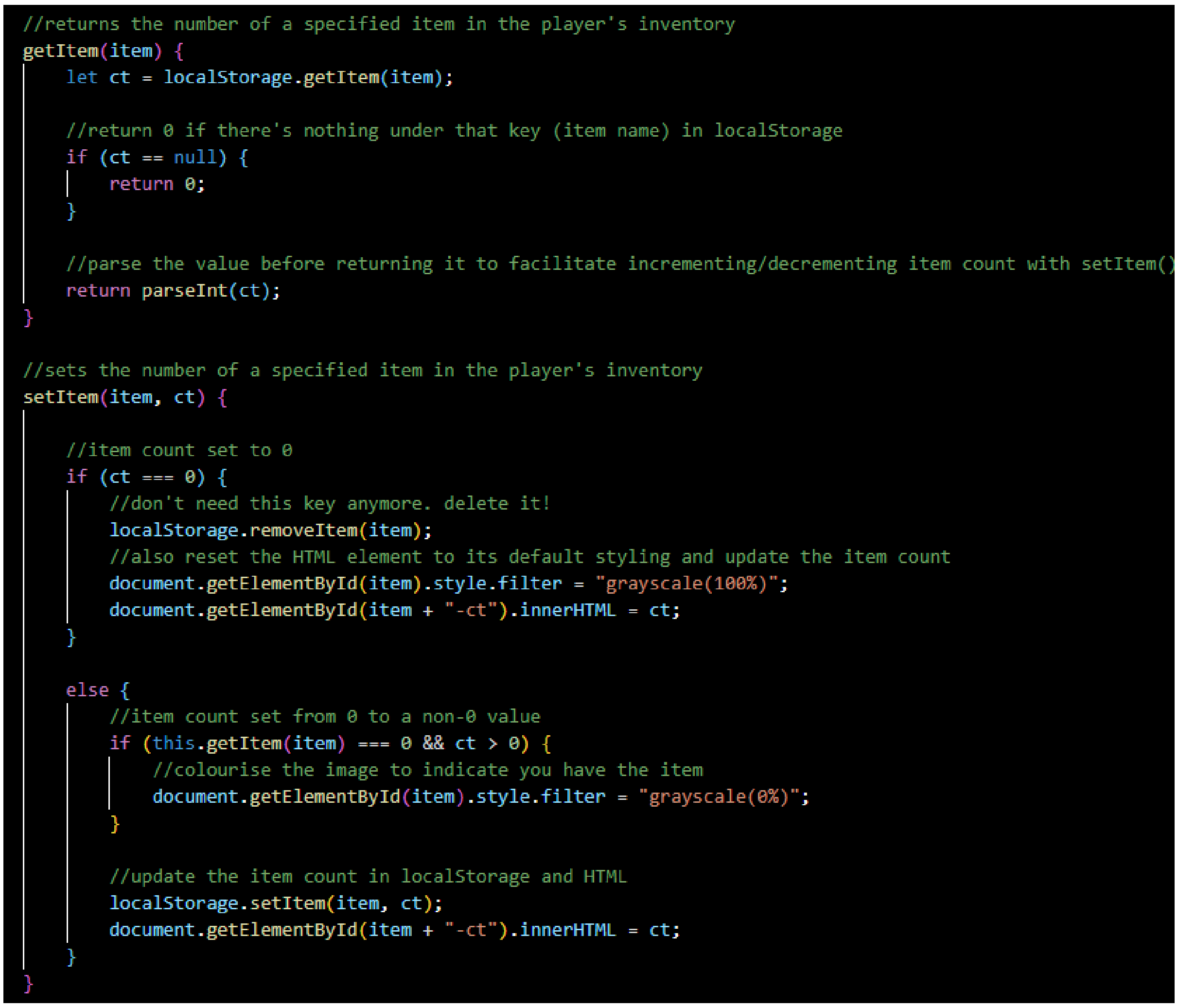
Biomes and structure chances are characteristics determined on a per-seed basis. While houses can appear in half of all seeds, deserts will only appear one tenth of the time. The Tile class manages this alongside most of the actual work managed by WaveFunctionCollapseManager, including collapsing to the best possibility, changing Tile identity during quality passes, and determining invalid neighbours.



```
//checks what we can do with the target tile
//returns true if the tile can be walked on, false if not
//i deeply apologize for how big the switch statement you're about to read is
interact(targetTile) {

  //reminder that Tile.possibilities[0] is the string identity of a collapsed tile
  switch (targetTile.possibilities[0]) {

    //if nothing is unique about this tile, you can walk on it and nothing happens
    default:
      return true;

    //similarly boring tiles that the player can never interact with or walk through
    case "Wall":
    case "PyramidWall":
    case "TowerWall":
    case "TowerStairsUp":
    case "BoatWheel":
    case "Table":
    case "Snowman":
    case "TotemComplete":
      return false;

    //pharoah interactions
    case "Sarcophagus":
      //pharoahs love idols and will take all of your idols before trading for ore
      if (this.inventory.getItem("idol") > 0) {
        this.inventory.setItem("idol", parseInt(this.inventory.getItem("idol")) - 1);
        this.inventory.setItem("orb", parseInt(this.inventory.getItem("orb")) + 10);
      }

      //no idols? fiiine, guess i'll take your ore.
      else if (this.inventory.getItem("ore") > 0) {
        this.inventory.setItem("ore", parseInt(this.inventory.getItem("ore")) - 1);
        this.inventory.setItem("orb", parseInt(this.inventory.getItem("orb")) + 1);
      }
      return false;

    //chop down trees
    case "Tree":
      targetTile.collapse("LogGrass");
      return false;
```

Player.interact()

The Player class manages player input and interactions with Tiles. When the player tries to move in a direction, Player determines whether they should wrap around to a new screen in original *The Legend of Zelda* fashion, walk onto the Tile, collapse the Tile to a new identity, collect an item, and so on.

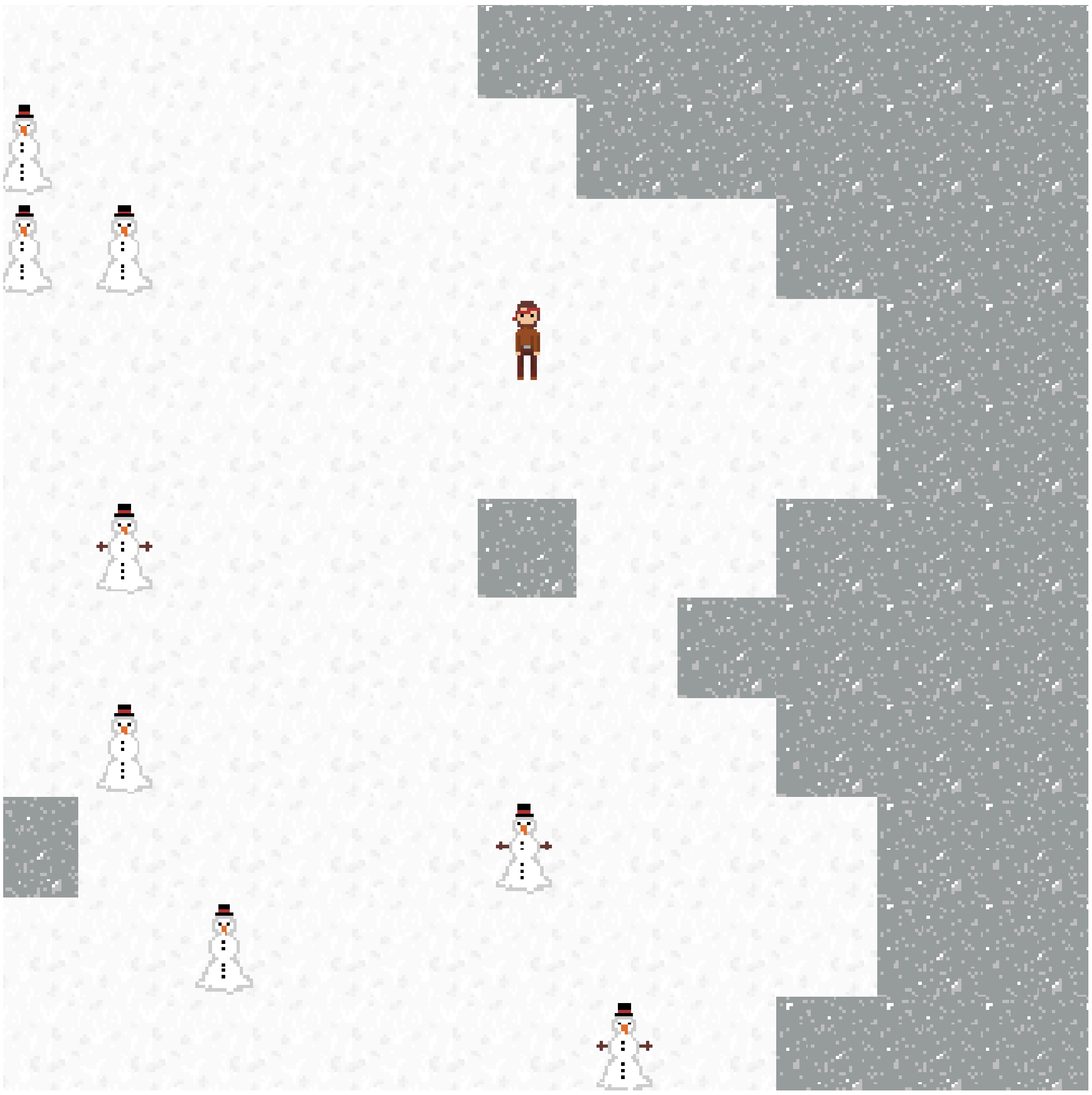


```
//returns the number of a specified item in the player's inventory
getItem(item) {
    let ct = localStorage.getItem(item);

    //return 0 if there's nothing under that key (item name) in localStorage
    if (ct == null) {
        return 0;
    }

    //parse the value before returning it to facilitate incrementing/decrementing item count with setItem()
    return parseInt(ct);
}

//sets the number of a specified item in the player's inventory
setItem(item, ct) {

    //item count set to 0
    if (ct === 0) {
        //don't need this key anymore. delete it!
        localStorage.removeItem(item);
        //also reset the HTML element to its default styling and update the item count
        document.getElementById(item).style.filter = "grayscale(100%)";
        document.getElementById(item + "-ct").innerHTML = ct;
    }

    else {
        //item count set from 0 to a non-0 value
        if (this.getItem(item) === 0 && ct > 0) {
            //colourise the image to indicate you have the item
            document.getElementById(item).style.filter = "grayscale(0%)";
        }

        //update the item count in localStorage and HTML
        localStorage.setItem(item, ct);
        document.getElementById(item + "-ct").innerHTML = ct;
    }
}
```

Inv.getItem() and Inv.setItem()

Players can collect and use items, which are saved in localStorage. Using localStorage allows players to swap between the game and information tabs without losing their hard-earned items. The Inv class acts as an “inventory” and facilitates interaction between the player and localStorage. Before items are collected, they are greyed out on the sidebar. This demonstrates that the items can be found but are not currently available. Once the player begins collecting items, Inv colourises the image and increases their count in localStorage and HTML. In-game, most items are collected and used in a lock and key method. Picks allow the player to mine a single mountain and obtain one piece of ore, which can then be used to craft a shovel and so on.

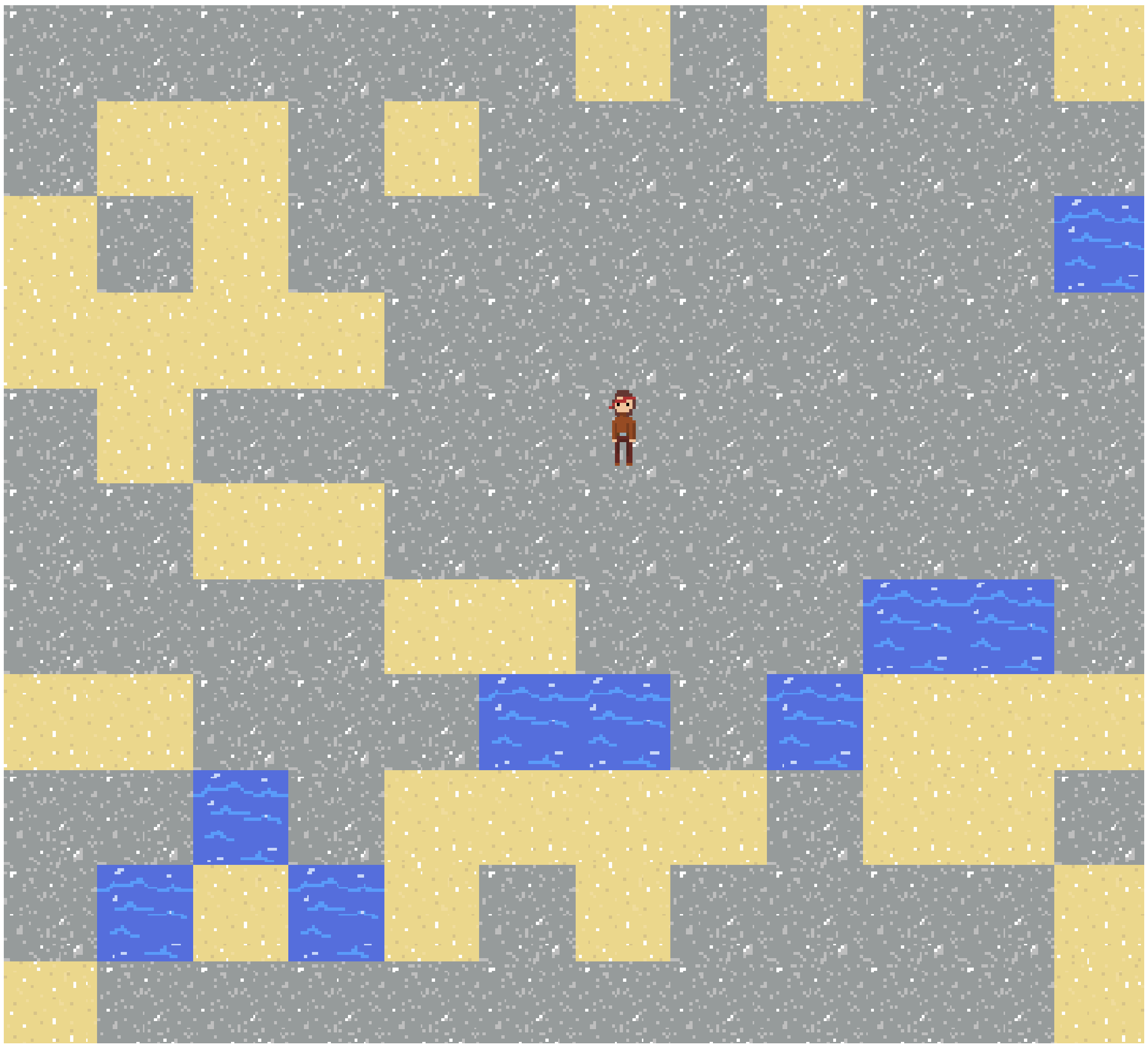
Critique



Snowmen

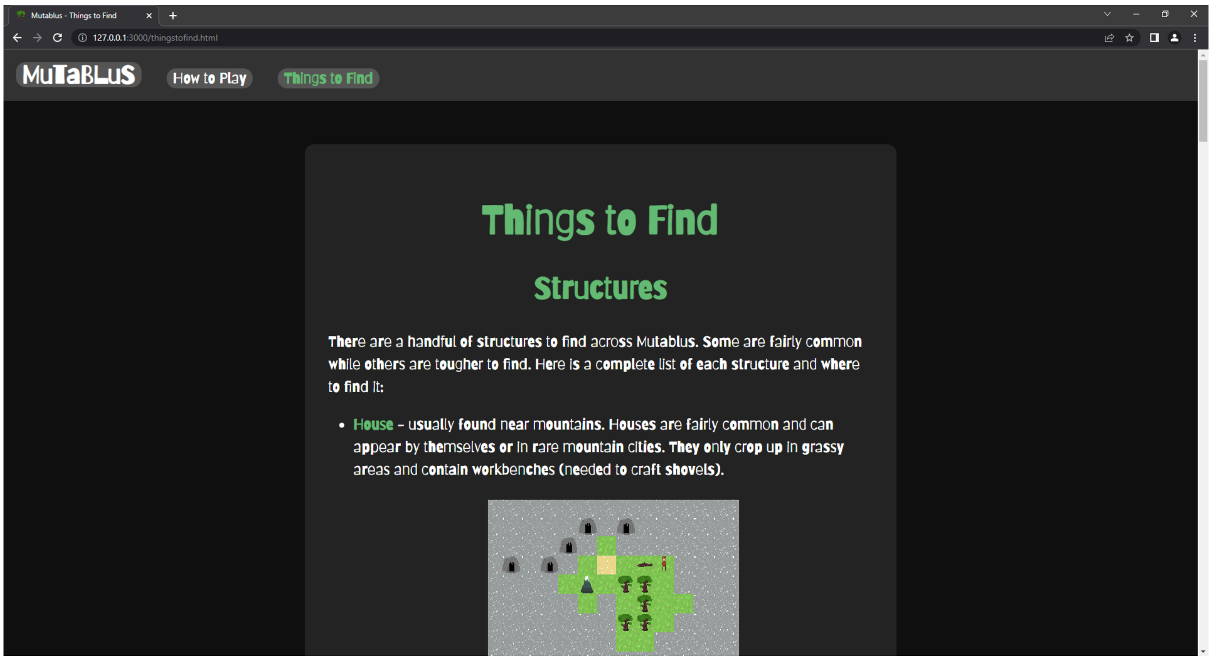
For all of my work on *Mutablus*, I cannot help but feel it is a simplified *Dwarf Fortress* clone. While the game includes the basics of an engaging world, it still needs many more systems and much more content to engage audiences. Deeper inventory management, more item functionality, other life in the world, and survival systems like hunger and warmth would boost replayability and retention time. The current screen organisation leaves much room for improvement as nearly half of the screen is blank space; this also goes for the information tabs which I initially designed with portrait orientation support in mind.

Outside of a lack of content, I do not see any glaring usability issues. However, I recognise I have set overly high expectations of myself for a procedurally generated web game. The user interface is very straightforward and should not confuse the player. Item management is surprisingly rewarding and complements exploration nicely. The inventory is easy to read and clearly communicates what items the player has. The procedural terrain generation is fun to explore and still surprises me weeks after its initial creation.



Strange biome combination

This project’s greatest achievement is implementing a reverse-engineered WFC algorithm without reading Gumin’s original code. This algorithm can generate strange, unique terrain without using octaves upon octaves of Perlin noise. I have been interested in WFC since before starting university, so finally implementing my own rendition was very rewarding. However, it is still very inefficient and needs refactoring. My WFC algorithm does not deterministically generate the same terrain from the same seed. It also does not fully propagate changes across the whole grid, though I decided this was for the best. It reduces the amount of work the algorithm has to do and, more importantly, allows the algorithm to occasionally violate its own rules to create rare, interesting combinations of tiles.



Things to Find

Structures

There are a handful of structures to find across Mutablus. Some are fairly common while others are tougher to find. Here is a complete list of each structure and where to find it:

- House - usually found near mountains. Houses are fairly common and can appear by themselves or in rare mountain cities. They only crop up in grassy areas and contain workbenches (needed to craft shovels).

Lots of free space in the margins

In the future, I would love to add mobile support as I have laid the foundation for a portrait orientation mode throughout development. I would like to add an option for a wider screen with more tiles for desktop users, but this would give them an undeniable advantage over players on smaller screens.

While I am pleased with my final result, *Mutablus* has not been developed or polished to satisfaction. It meets its goals on the code and accessibility fronts but the gameplay loop is severely lacking. The game needs more content in the form of biomes, structures, and items. Systems like Tile image drawing and Player interactions could be refactored for better performance. I am satisfied with the content I did add and believe every point of interest is recognisable and deserving of its place in the game.

Word count: 2035

References

Atari, Inc. (1980). *Adventure*. [Video game].

Bay 12 Games. (2006). *Dwarf Fortress*. [Video game].

Donald, M. (2020). *Wave Function Collapse*. [Computer algorithm]. Available at https://bolddunkley.itch.io/wave-function-collapse [Accessed 3 April 2023].

Gumin, M. (2016). *Wave Function Collapse*. [Computer algorithm]. Available at https://github.com/mxgmn/WaveFunctionCollapse [Accessed 3 April 2023].

Mojang. (2011). *Minecraft*. [Video game].

Nintendo. (1986). *The Legend of Zelda*. [Video game].

Origin Systems. (1983). *Ultima III: Exodus*. [Video game].

Plausible Concept. (2018). *Bad North*. [Video game].

Processing Foundation. (2016). *p5.js*. [JavaScript plugin]. Available at https://p5js.org/ [Accessed 1 April 2023].

Watt Designs. (2023). *A new way to generate worlds*. [Video]. Available at: https://youtu.be/dFYMOzoSDNE?t=49 [Accessed 2 April 2023].