PRINCÉTEMON WRITTEN REPORT

Arnav Kumar, Stephanie Yen, Kirsten Pardo
COS426 – Prof. Felix Heide
Spring 2022

ABSTRACT

Princetémon is a Princeton-themed top-down role-playing video game (RPG). Inspired by the plot-driven and interactive nature of Pokémon, as well as landmarks of Princeton's campus, Princetémon consists of several scenes and mini-games that are navigated by an individual player, who is a student at Princeton University. In order to enter Frist Campus Center, which has been locked by Christopher Nolan's filming crew, the student must complete mini-games to collect props necessary for entrance from various locations on campus, such as Poe Field (where they have to dodge construction debris), The Street (where they have to learn a dance choreography), and Prospect Garden (where they avoid water piranhas that have infested the fountain). Visually, the game has a retro 8-bit pixelated style.

INTRODUCTION

Originally, we knew we wanted to make a fun and smooth game, and we knew we wanted it to be Princeton-themed. Along with some inspiration from games such as Pokémon and Stardew Valley, we realized that a top-down role-playing game (RPG) was a great way to combine these two goals. A top-down role-playing game is a game where the user controls and takes the role of a character in the game, but the entire game is viewed from an elevated position where you can see the character. Both Pokémon and Stardew Valley follow this blueprint, and we decided to make a game similar in general mechanics to these, but with an added Princeton flavor. In order to make a game like this, we needed our own storyline, which ended up being inspired by a surprise visit by Christopher Nolan to Princeton to film. In addition, we figured we needed there to be some sort of challenge to the game. We were inspired by the challenges in the aforementioned games– in Pokémon, collecting Pokémon and battling other leaders is the challenge, while in Stardew Valley, collecting farm commodities was similarly the main challenge. In Princetémon, the challenge is completing three minigames and collecting the necessary rewards to achieve the game's ultimate goal. Lastly, there was the issue of actually implementing a walkable pixelated map. In this area, we were inspired by the previous COS426 project Cave Climber to use a tile-based map system for the purpose of efficiency. However, their use was to make a tile-based platformer, while we created a tile-based top-down game, meaning we had to adjust that concept to fit our idea. With these inspirations, our ideas for a Princeton-themed top-down RPG became much clearer. Our minimum viable product (MVP) was three scenes (Frist, Poe Field, and Prospect Street) and two minigames, with stretch goals to add a fourth scene, a third minigame, and a storyline with dialogue.

In our current product, we have five scenes (Frist, Poe Field, Prospect Street, Nassau Hall, and Prospect Gardens) and three minigames, as well as a storyline with dialogue. The game starts off in Frist South Lawn, where the user must talk to a student in front of Frist Campus Center. The student tells the user that Frist is locked due to the filming of Floppenheimer by Christopher Nolan, which upsets the user since he left his signed picture of Professor Felix Heide inside. The student says that if the user can collect three oddly specific rewards, perhaps Christopher Nolan will let him inside. The user then travels to the other four locations, three of which (Poe Field, Prospect Street, and Prospect Gardens) have minigames initiated by dialogues with characters. The user can replay the minigame as many times as possible until victorious. At any point, the user can also check a map to see which scene they are currently in, check their rewards progress, move the camera, or move the sprite to traverse a scene or switch scenes. Once the user wins all three, they can travel back to Frist and walk in. The final scene is a small scene inside Frist, where the user talks to Christopher Nolan and wins the game. We will discuss the implementations for all this in the next section.

METHODOLOGY

1. Designing and Obtaining Tilesets

The first thing we did was figure out how to get usable pixel art. Initially, we wanted to hand design every tile used. However, we quickly realized this was infeasible, as we need lots of different tiles. Therefore, many of the more basic tiles came from open-source tilesets on itch.io, such as grass, trees, paths, roads, some sprites, etc. This helped us create bigger maps in shorter amounts of time while preserving uniformity, as many of the tiles were designed to be used together. It also aided in terms of efficiency, as the larger the tileset the easier the import. While open-source tilesets were extremely helpful when creating the scenes, we could not rely on them completely since we were designing the Princeton campus, which was not available in open-source tilesets. For this, we used Piskel, a free online pixel-art editor, to design our own custom pixel arts for things such as Princeton buildings, main sprites, and other more scene-specific items. We then imported these images or combined them into custom tilesets. With these, we were able to effectively design large custom maps without having to start completely from scratch.

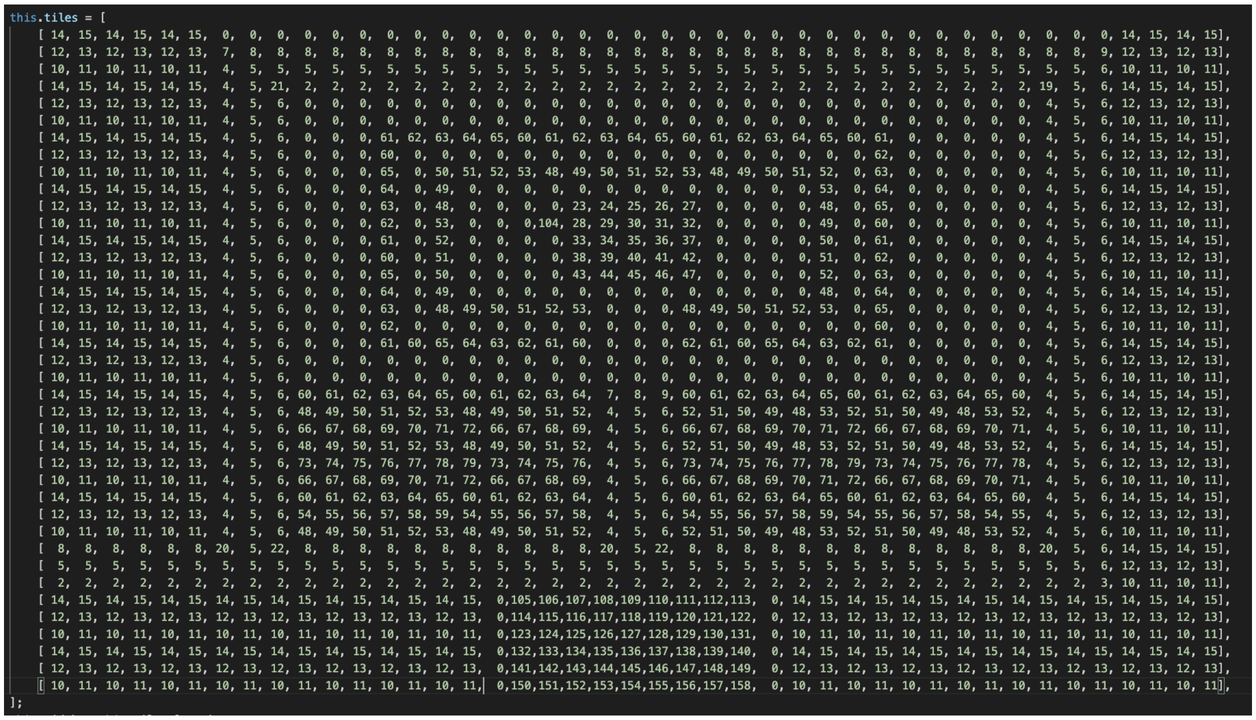
Example of custom pixel art for Nassau Hall, created with Piskel

2. Creating Tile-Based Maps With ThreeJS

With help from an online HTML5 tutorial and Cave Climber, we decided to use the popular concept of tilemaps in order to create the scenes. To do so, we read in images or tilesets and defined the amount of tiles in the x- and y-directions of the image. We then specified the location of the tile we wanted and created a ThreeJS Texture for the given tile. Each Texture tile was the same size on the screen, so for objects that we wanted to be bigger, such as buildings, we read in multiple tiles and placed them accordingly on the map in order to create the perception of a bigger object. For example, the Frist Campus Center building took a total of 88 tiles to create. To store all the different tiles, we assigned indices to each and put them into a HashMap. This was the method used to get the different Texture tiles, but there was still the issue of combining the Texture tiles to create full scenes.

To create full scenes, we created a 2D array storing the index of the tile at each location. Then, we iterated through it and created a ThreeJS Sprite with the texture of the index of the tile. This Sprite was placed at the given location and added to the scene. For some tiles, the background was transparent, but in a top-down RPG, there should be no transparency. Therefore, for certain tiles, we defined a tile that should be placed in the background in order to remove transparency.

To create the perception of a top-down game in a 3D framework, we used a Perspective Camera and set the position x- and y-coordinates to the same coordinates as the point the camera looked at. We set the position z-coordinate to a positive number (1.6) and set it to look at z = 0, literally creating an aerial view. The Perspective Camera was used in order to avoid resizing tiles when resizing the window, which occurred with the Orthographic Camera. This transformation from 3D to 2D also allowed for things such as background tiles, as we could set the z-coordinate to slightly less than 0 to give the appearance of being under.



```
this.tiles = [
    [ 14, 15, 14, 15, 14, 15,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 14, 15, 14, 15],
    [ 12, 13, 12, 13, 12, 13,  7,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  9, 12, 13, 12, 13],
    [ 10, 11, 10, 11, 10, 11,  4,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  6, 10, 11, 10, 11],
    [ 14, 15, 14, 15, 14, 15,  4,  5, 21,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2, 19,  5,  6, 14, 15, 14, 15],
    [ 12, 13, 12, 13, 12, 13,  4,  5,  6,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  4,  5,  6, 12, 13, 12, 13],
    [ 10, 11, 10, 11, 10, 11,  4,  5,  6,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  4,  5,  6, 10, 11, 10, 11],
    [ 14, 15, 14, 15, 14, 15,  4,  5,  6,  0,  0,  0, 61, 62, 63, 64, 65, 60, 61, 62, 63, 64, 65, 60, 61, 62, 63, 64, 65, 60, 61,  0,  0,  0,  0,  0,  4,  5,  6, 14, 15, 14, 15],
    [ 12, 13, 12, 13, 12, 13,  4,  5,  6,  0,  0,  0, 60,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 62,  0,  0,  0,  0,  0,  4,  5,  6, 12, 13, 12, 13],
    [ 10, 11, 10, 11, 10, 11,  4,  5,  6,  0,  0,  0, 65,  0, 50, 51, 52, 53, 48, 49, 50, 51, 52, 53, 48, 49, 50, 51, 52,  0, 63,  0,  0,  0,  0,  0,  4,  5,  6, 10, 11, 10, 11],
    [ 14, 15, 14, 15, 14, 15,  4,  5,  6,  0,  0,  0, 64,  0, 49,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 53,  0, 64,  0,  0,  0,  0,  0,  4,  5,  6, 14, 15, 14, 15],
    [ 12, 13, 12, 13, 12, 13,  4,  5,  6,  0,  0,  0, 63,  0, 48,  0,  0,  0,  0, 23, 24, 25, 26, 27,  0,  0,  0,  0, 48,  0, 65,  0,  0,  0,  0,  0,  4,  5,  6, 12, 13, 12, 13],
    [ 10, 11, 10, 11, 10, 11,  4,  5,  6,  0,  0,  0, 62,  0, 53,  0,  0,  0,104, 28, 29, 30, 31, 32,  0,  0,  0,  0, 49,  0, 60,  0,  0,  0,  0,  0,  4,  5,  6, 10, 11, 10, 11],
    [ 14, 15, 14, 15, 14, 15,  4,  5,  6,  0,  0,  0, 61,  0, 52,  0,  0,  0,  0, 33, 34, 35, 36, 37,  0,  0,  0,  0, 50,  0, 61,  0,  0,  0,  0,  0,  4,  5,  6, 14, 15, 14, 15],
    [ 12, 13, 12, 13, 12, 13,  4,  5,  6,  0,  0,  0, 60,  0, 51,  0,  0,  0,  0, 38, 39, 40, 41, 42,  0,  0,  0,  0, 51,  0, 62,  0,  0,  0,  0,  0,  4,  5,  6, 12, 13, 12, 13],
    [ 10, 11, 10, 11, 10, 11,  4,  5,  6,  0,  0,  0, 65,  0, 50,  0,  0,  0,  0, 43, 44, 45, 46, 47,  0,  0,  0,  0, 52,  0, 63,  0,  0,  0,  0,  0,  4,  5,  6, 10, 11, 10, 11],
    [ 14, 15, 14, 15, 14, 15,  4,  5,  6,  0,  0,  0, 64,  0, 49,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 48,  0, 64,  0,  0,  0,  0,  0,  4,  5,  6, 14, 15, 14, 15],
    [ 12, 13, 12, 13, 12, 13,  4,  5,  6,  0,  0,  0, 63,  0, 48, 49, 50, 51, 52, 53,  0,  0,  0, 48, 49, 50, 51, 52, 53,  0, 65,  0,  0,  0,  0,  0,  4,  5,  6, 12, 13, 12, 13],
    [ 10, 11, 10, 11, 10, 11,  4,  5,  6,  0,  0,  0, 62,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 60,  0,  0,  0,  0,  0,  4,  5,  6, 10, 11, 10, 11],
    [ 14, 15, 14, 15, 14, 15,  4,  5,  6,  0,  0,  0, 61, 60, 65, 64, 63, 62, 61, 60,  0,  0,  0, 62, 61, 60, 65, 64, 63, 62, 61,  0,  0,  0,  0,  0,  4,  5,  6, 14, 15, 14, 15],
    [ 12, 13, 12, 13, 12, 13,  4,  5,  6,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  4,  5,  6, 12, 13, 12, 13],
    [ 10, 11, 10, 11, 10, 11,  4,  5,  6,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  4,  5,  6, 10, 11, 10, 11],
    [ 14, 15, 14, 15, 14, 15,  4,  5,  6, 60, 61, 62, 63, 64, 65, 60, 61, 62, 63, 64,  7,  8,  9, 60, 61, 62, 63, 64, 65, 60, 61, 62, 63, 64, 65, 60,  4,  5,  6, 14, 15, 14, 15],
    [ 12, 13, 12, 13, 12, 13,  4,  5,  6, 48, 49, 50, 51, 52, 53, 48, 49, 50, 51, 52,  4,  5,  6, 52, 51, 50, 49, 48, 53, 52, 51, 50, 49, 48, 53, 52,  4,  5,  6, 12, 13, 12, 13],
    [ 10, 11, 10, 11, 10, 11,  4,  5,  6, 66, 67, 68, 69, 70, 71, 72, 66, 67, 68, 69,  4,  5,  6, 66, 67, 68, 69, 70, 71, 72, 66, 67, 68, 69, 70, 71,  4,  5,  6, 10, 11, 10, 11],
    [ 14, 15, 14, 15, 14, 15,  4,  5,  6, 48, 49, 50, 51, 52, 53, 48, 49, 50, 51, 52,  4,  5,  6, 52, 51, 50, 49, 48, 53, 52, 51, 50, 49, 48, 53, 52,  4,  5,  6, 14, 15, 14, 15],
    [ 12, 13, 12, 13, 12, 13,  4,  5,  6, 73, 74, 75, 76, 77, 78, 79, 73, 74, 75, 76,  4,  5,  6, 73, 74, 75, 76, 77, 78, 79, 73, 74, 75, 76, 77, 78,  4,  5,  6, 12, 13, 12, 13],
    [ 10, 11, 10, 11, 10, 11,  4,  5,  6, 66, 67, 68, 69, 70, 71, 72, 66, 67, 68, 69,  4,  5,  6, 66, 67, 68, 69, 70, 71, 72, 66, 67, 68, 69, 70, 71,  4,  5,  6, 10, 11, 10, 11],
    [ 14, 15, 14, 15, 14, 15,  4,  5,  6, 60, 61, 62, 63, 64, 65, 60, 61, 62, 63, 64,  4,  5,  6, 60, 61, 62, 63, 64, 65, 60, 61, 62, 63, 64, 65, 60,  4,  5,  6, 14, 15, 14, 15],
    [ 12, 13, 12, 13, 12, 13,  4,  5,  6, 54, 55, 56, 57, 58, 59, 54, 55, 56, 57, 58,  4,  5,  6, 54, 55, 56, 57, 58, 59, 54, 55, 56, 57, 58, 54, 55,  4,  5,  6, 12, 13, 12, 13],
    [ 10, 11, 10, 11, 10, 11,  4,  5,  6, 48, 49, 50, 51, 52, 53, 48, 49, 50, 51, 52,  4,  5,  6, 52, 51, 50, 49, 48, 53, 52, 51, 50, 49, 48, 53, 52,  4,  5,  6, 10, 11, 10, 11],
    [  8,  8,  8,  8,  8,  8, 20,  5, 22,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8, 20,  5, 22,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8, 20,  5,  6, 14, 15, 14, 15],
    [  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  6, 12, 13, 12, 13],
    [  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  3, 10, 11, 10, 11],
    [ 14, 15, 14, 15, 14, 15, 14, 15, 14, 15, 14, 15, 14, 15, 14, 15,  0,105,106,107,108,109,110,111,112,113,  0, 14, 15, 14, 15, 14, 15, 14, 15, 14, 15, 14, 15, 14, 15, 14, 15],
    [ 12, 13, 12, 13, 12, 13, 12, 13, 12, 13, 12, 13, 12, 13, 12, 13,  0,114,115,116,117,118,119,120,121,122,  0, 12, 13, 12, 13, 12, 13, 12, 13, 12, 13, 12, 13, 12, 13, 12, 13],
    [ 10, 11, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11,  0,123,124,125,126,127,128,129,130,131,  0, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11],
    [ 14, 15, 14, 15, 14, 15, 14, 15, 14, 15, 14, 15, 14, 15, 14, 15,  0,132,133,134,135,136,137,138,139,140,  0, 14, 15, 14, 15, 14, 15, 14, 15, 14, 15, 14, 15, 14, 15, 14, 15],
    [ 12, 13, 12, 13, 12, 13, 12, 13, 12, 13, 12, 13, 12, 13, 12, 13,  0,141,142,143,144,145,146,147,148,149,  0, 12, 13, 12, 13, 12, 13, 12, 13, 12, 13, 12, 13, 12, 13, 12, 13],
    [ 10, 11, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11,  0,150,151,152,153,154,155,156,157,158,  0, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11, 10, 11],
];
```

Example of a tilemap used in the game: each number corresponds to a different ThreeJS Texture tile. Using a HashSet, we were able to note which of these indices a player could walk on and which changed scenes.

3. Creating the Player and Movement

To create the player sprite, we followed a similar technique as with creating tile-based maps. We first designed a 4x3 tileset, where each row was a direction of walking (left, right, down, up) and each column was either standing straight, left foot forward, or right foot forward. After reading and storing these into Textures in a HashMap, we placed the sprite into the given scene at the same point the camera looked at.

To move the character, we created a keydown event handler and listened for the arrow keys. Depending on the arrow key pressed, the player sprite's position would move 0.5 in the direction, and the camera would move along with the player to keep the player in the center of the screen. The real difficulty was in animating the player's movement. To do this, we created a cycle loop of [0, 1, 0, 2] and set a currentIndex for this loop, initialized at 0. Every time an arrow key was pressed, the currentIndex increased by 1, or if at 3, was set back to 0. This represented whether the sprite was standing straight (0), walking left foot forward (1), or walking right foot forward (2), meaning the cycle loop was stand, left foot, stand, right foot (0, 1, 0, 2) and was executed incrementally as arrow keys were pressed. In order to make this visible, we had to remove the sprite object from the scene, update the animation texture for the sprite, and add the sprite back each time an arrow key was pressed. This created the perception that the player's legs were moving when walking in any direction.

Another thing we had to account for when creating player movement was which tiles the player could and could not step on. For example, the player should be allowed to walk along grass or a path, but not along a tree tile. To accomplish this, we created a HashSet storing the indices of all tiles that were walkable. Then, when adjusting the player sprite's location, we checked to see if the tile being moved to was in the walkable HashSet. If yes, we moved the player. If not, the player did nothing.

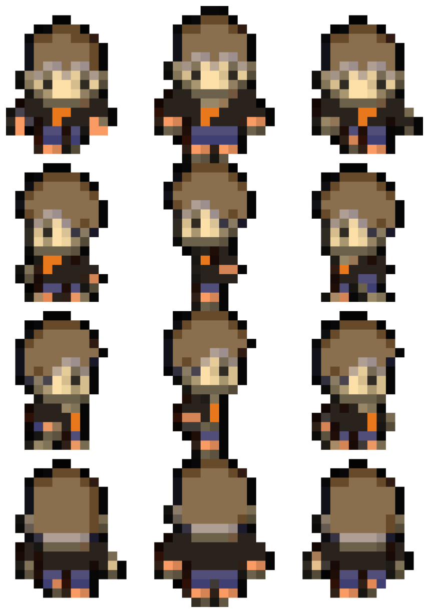


Image of the sprite sheet used for the main player: there are four columns, one for each walking direction, and three steps for each direction. This allowed us to use the cycle loop to animate the walking.

4. World Navigation and Scene Switching

Each of the five different locations was designed using the tile-based methods, but in order for the game to function properly, we had to add logic for switching between locations. To deal with this, we created a separate Scenes class to hold all the different scenes of the game in a dictionary. Within this, we added a method to switch scenes. This method took in a key and set the current scene to the scene at the new key. In addition, we put the renderer inside this Scenes class. Within the animation loop, we changed the renderer to render the current scene rather than a specific one, which made it so that the current scene was always the one being shown.

In order to switch between scenes, we used a similar solution to the walkable HashSet by creating a scene changer HashSet. This HashSet contained all the indices of tiles that would switch the scene if the user pressed the correct direction. For example, in Frist, we added the path tiles to the scene changer HashSet, and checked at the edges of the screen if the current tile was a path tile. Then, if the player was at the left edge on a path tile and the left arrow was pressed, it would switch to Prospect Street, and so on for the other edges.

It is also important to note that each scene had a separate player object within it. Therefore, when switching scenes, what really happens is the current scene changes to the new scene, and the new scene has a player object defined and initialized in a certain position, while the old scene's player remains at the edge of the old scene.

5. Minigames

For the three rewards to beat the game, there were three separate minigames, one in Poe Field, one in Prospect Street, and one in Prospect Gardens. Each game had similar logic, although the games each had their own special qualities.

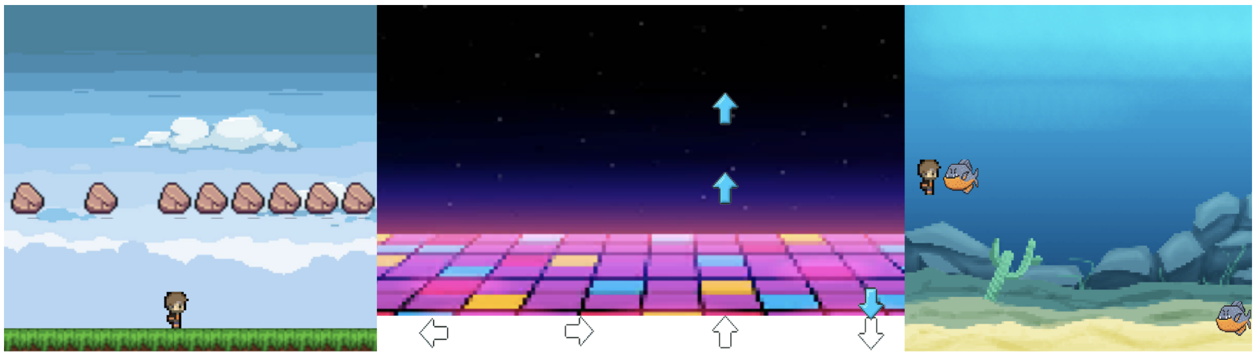
The Poe Field minigame involves only left and right player movement, while rocks fall from the sky. The goal is to avoid being hit by the rocks long enough. There are 17 rocks and 17 positions for the user to walk between, with one rock in each spot. For the sake of uniqueness, rather than falling at random times, the rocks fall in troves. The first stage has only one rock fall. The second has three rocks fall, the third has five, and so on until 15 rocks fall leaving only two safe spots to stand. To code this rock falling logic, we first set the position high enough to be out of view. Then, using setInterval(), we kept track of how many rocks should fall and generated that many unique random positions between 0 and 16. For each of these random positions, we called a helper function rockFall() to make the rock at that position start its descent. Within rockFall(), we call setInterval again. This time, we keep a time variable that counts until it hits a limit, which was set to when the rocks hit the bottom of the screen. In each iteration of the interval, we drop the rock called upon by 0.5, and this keeps iterating until the falling rocks reach the bottom of the screen. Then, in the original setInterval, we updated the amount of rocks, regenerated random numbers, and called rockFall() again in the next iteration until the game ended. To check if a rock hit the player, we added logic within rockFall() that checked if the rock's y-position was in between the range of [0.55, 1.45] (which is the player's y-position range) and checked if the player's x-position was at the index of the falling rock. If so, the player was getting hit and the game ended. If not, we set the rocks back to the original position and waited for them to fall again. This was the general method used to create the rock game.

The Prospect Street minigame requires all four arrow keys. It involves arrows falling from the sky, and the user must press the corresponding arrows at the right times in order to win, similar to the game Dance Dance Revolution. This game once again used a setInterval() with a time variable to determine when the game should end. In addition, there were 8 arrow sprites used, 2 for each direction, placed above the screen. This time, however, the arrow sprites were to fall randomly rather than in troves, which meant changing up the logic. In order to accomplish this, we kept a separate list of speeds, one for each of the 8 arrow sprites, which kept track of whether the arrow should be moving or not. Then, during each interval iteration, we got a random index, and if the arrow was not moving, we set it to move to the current speed of the game, which kept increasing as time went on. For the falling logic of the arrows, we simply decreased the position by the speed of the arrow, which was 0 if the arrow was not selected to fall yet. Our iterations repeated much more often than in the rock game, which created the perception that the arrows were falling continuously rather than being dropped a step at a time. With the arrow logic complete, we added the logic to see if the user pressed the key at the right time. Within the arrow key event handlers, we checked if either of the two arrow sprites corresponding to the key pressed had a y-position less than 1, in which case we sent them back to the original y-position. The user lost if any arrow sprite made it past y-position -0.8, and the user won if the time reached a certain limit.

The Prospect Garden minigame was a bit of a combination between the previous two games. In this game, there are only up and down movements, and piranhas come in from the right randomly. The goal is to avoid the piranhas and grab the coin once it arrives. To make this, all the y-coordinates were changed into x-coordinates to make the piranhas come in from the right rather than the top. The piranhas were coded exactly as the arrows in the Prospect Street game, and the collision detection was the same as in the Poe

Field game. The only big difference was a coin was randomly spawned during certain time ranges, and once the player collided with the coin, they won the game.

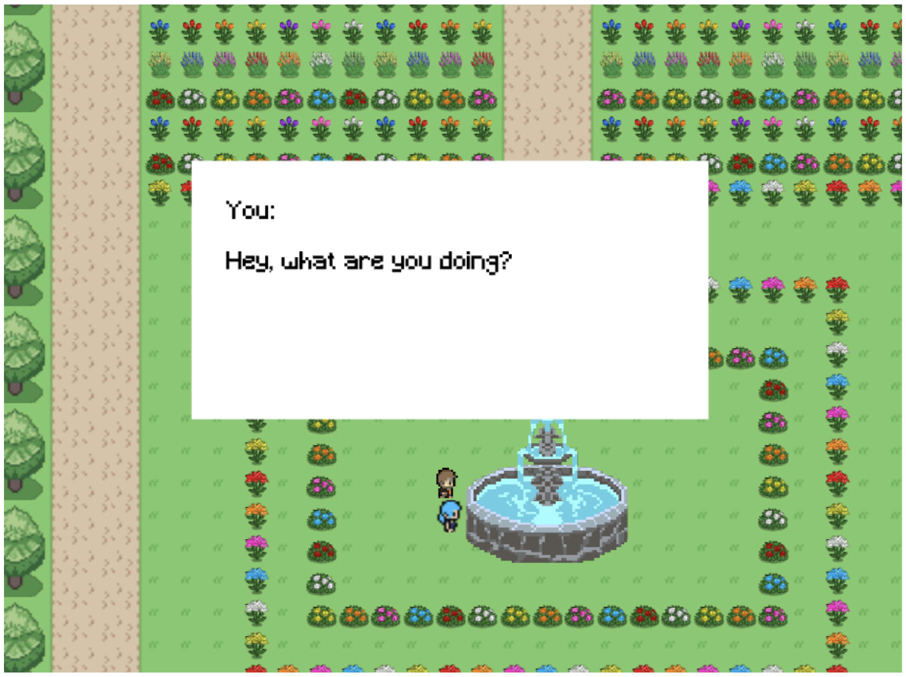
For each of the games, it was important to clear the setInterval no matter the outcome so the user could play the games again from the start as many times as they wanted. In addition, each of the games requires some user input (spacebar or movement) to start, as without this, the setInterval would run immediately as the overall game loads up and the minigames would be empty.



Images of gameplay during the Poe, Street, and Gardens minigames

6. Dialogue

To implement dialogue, we needed a way to add and remove text and shapes just with the press of the spacebar. To do this, first we removed the current event handlers while the dialogue was occurring to avoid any bugs, and we added them back once the dialogue was completed. Then, in each scene, we added an instance variable for the current text and for the box behind it. The box was created using ThreeJS BoxGeometry and the text was written with ThreeJS TextGeometry, each assigned to the instance variable. This was important as we could now access the box and text within the font loading through the Scenes class. On each press of space, a new dialogue was coded in and showed up by removing the previous text, assigning the new text, and adding it back into the scene. At the end of the dialogue, the text and box were both removed entirely, and the old event handlers were added back. Lastly, we adjusted the camera at the start of each dialogue to make sure the dialogue was viewable. This created the feeling of a speech bubble and custom dialogues of any length.


You:

Hey, what are you doing?

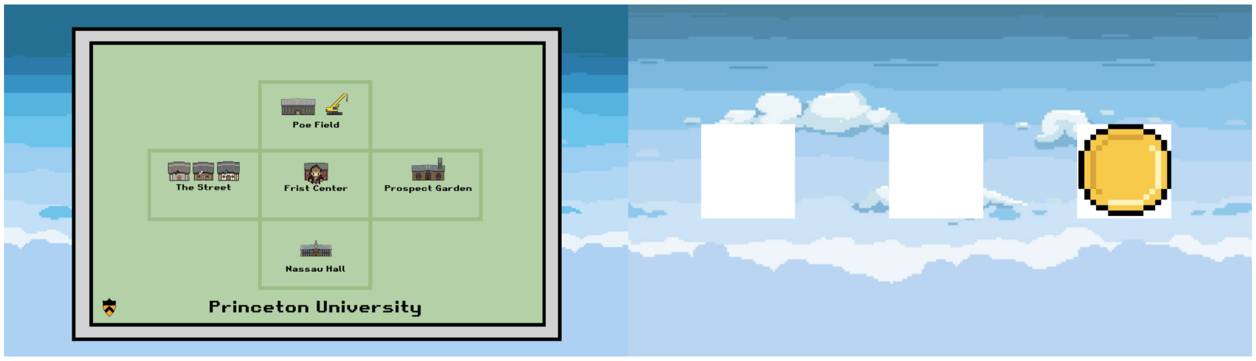
Example of design used for dialogue boxes when talking with other sprites

7. Miscellaneous Event Handlers

To make the game more playable and user-friendly, several other event handlers were added in. The first was the maps. If the user presses ‘m’, a map shows up with a general indication of which scene the player is in. This was achieved through a keydown event handler that created a new Maps object and added it to the Scenes class. The Maps object was simply an image that was determined by where the Maps object was created from by passing in a location to the Maps constructor. Since we passed in a location to the constructor, we had to create a new Maps object for each press of ‘m’, and this allowed us to press ‘m’ again to return to the location that was passed in the constructor.

A second event handler was the rewards progress page. It was created the same way as the maps, and a new Rewards object was created each time the user pressed ‘r’. This page was simply three tiles read in from a tileset. To check if a reward was obtained or not, we created another instance variable in the Scenes class that held whether each reward was collected. Then, every time ‘r’ was pressed, we created a new Rewards object with the information of which rewards had been collected from the Scenes class, and displayed the corresponding ones appropriately.

The last event handlers were the movement and zoom of the camera. The movement of the camera was done very similarly to the player movement and just reacted to ‘wasd’ key presses, but was simpler as it did not need to worry about walkable tiles. The zoom of the camera was even simpler, as we just switched the zoom attribute of the camera between two values depending on whether ‘z’ or ‘x’ was pressed.



Examples of the Map and Rewards scenes created each time 'm' and 'r' were pressed respectively

RESULTS

Overall, we have surpassed the expectations we had for our MVP– we have generated 2 additional scenes with Prospect Garden and Nassau Hall/Cannon Green, as well as a third fountain minigame in Prospect Garden. We also added dialogue boxes, and are working on developing the story and challenges.

Throughout the process, there was quite a bit of experimentation with the graphics and detail/quality in the sprites so we could create a cohesive visual style within the game (ex. using sprites with different dimensions was jarring due to blurriness of some sprites, using highly detailed sprites with a wider color palette vs simpler sprites that had less shadows/coloring/different outline styles did not look visually appealing, etc.) Alongside this, we got some initial user feedback from peers about how the world seemed sparse, so we experimented with different sprites in order to make the world seem more populated, sought out/are working on adding SFX in order to make the game feel upbeat, and will continue to add interaction/dialogue/NPCs into the game. We also noticed within the minigames, such as The Street dancing minigame, that pacing in the gameplay was a bit off (in terms of the arrow frequency coming down) so we adjusted rates of movement in that game and applied the concept to other minigames. All in all, our additions in response to the experimentation and feedback has helped improve the aesthetics and mechanics of the game, and we will be spending more time developing the world.

DISCUSSION + CONCLUSION

The approach we took to developing this game has been successful and promising, and we are satisfied in how we effectively surpassed our MVP goals, as well as how we have grown the game visually using our personalized sprites. Alongside this, we have gotten feedback from peers at how enjoyable the game's graphics, storyline, and minigames are and have picked up many skills quickly in terms of all the different game development components: graphic design, figuring out HTML5 and coding mechanics, and learning how to acquire all the different assets involved within a game or making them ourselves. This game and many 8-bit games, though seemingly simple, require many design assets to feel populated, which is something that we have learned through trial and error. We will use this experience going forward as we aim to improve the aesthetics of the game (like adding more sprites/NPCs/etc.) and the interactions with the environment (like being able to interact with landmarks to read more about them, adding sounds and ambience.)

More specifically, some features we want to improve or add moving forward are enhancing the walkability to allow for walking under some object tiles (such as street lights), adding specific sound effects and environment sounds for different scenes and games, and creating more animated structures that run constantly (like running water in the fountain or moving cars on the street). In addition, making the game more complex would likely add to the user experience, so adding more Princeton-themed scenes and minigames to complete would be another future step to consider. Lastly, adding a timer or death counter and creating a backend with a leaderboard could create a more competitive purpose for Princetémon.

CONTRIBUTIONS

Arnav and Stephanie laid out a lot of the groundwork for coding structure and gameplay/minigames, while Kirsten was responsible for designing/drawing all sprite assets and any additional pixel art (maps, title screens, etc.) throughout the game. More specifically, Arnav was in charge of the initial code setup, player mechanics, and minigames, while Stephanie was in charge of reading in tilesets, designing the tilemaps for the five different scenes, and dealing with scene switching. We worked together on the proposal/slides, written report, and will work together on the demo as well.

WORKS CITED

- Cave Climber: https://github.com/Derndeff/cave-climber/
- Cakery Bakery: https://github.com/cz10/thecakerybakery
- Piskel: https://www.piskelapp.com/
- Pixel tilesets
    - https://limezu.itch.io/serenevillage
    - https://emily2.itch.io/modern-city?download
    - https://xenophero.itch.io/rock-sprites
    - https://tilation.itch.io/16x16-small-indoor-tileset
    - https://route1rodent.itch.io/16x16-rpg-character-sprite-sheet
- Tutorials
    - ThreeJS: https://threejs.org/docs/
    - Tile-based HTML5 tutorial: https://www.creativebloq.com/html5/build-tile-based-html5-game-31410992
    - Animation tutorial: https://dev.to/martyhimmel/animating-sprite-sheets-with-javascript-ag3
- Image pixelator: https://pinetools.com/pixelate-effect-image
- Image background remover: https://www.remove.bg