Puzzled at Princeton

Danica Truong, Lauren Gardner, Warren Quan
COS 426 – Fall 2022

Abstract:

Inspired by games like Baba is You, we wanted to do a top-down puzzle game. The main mechanics would allow the player to push letters, allowing the user to input them to solve a provided riddle. Each puzzle is contained to its own room or level and you solve the puzzle to access the next room with a new puzzle. Lastly, we hoped to achieve a look similar to The Legend of Zelda: A Link Between Worlds, where the camera is in a top-down 2D perspective but has a skewed view to better display the 3D models.

Introduction

Eisgruber has locked you under Nassau Hall in a citation of academic mediocrity. To test whether you are still smart enough to stay at Princeton, Eisgruber has built yet another highly aspirational environment where you must finish all the puzzles laid out before you can escape back into the orange bubble. For our game, we programmed a screen control panel, multiple views, sound, collision detection, texture mapping, and view frustum culling through story cutscenes and puzzle gameplay.

Methodology

1. Character Design

One of the first things we tackled as a group was getting our 3D models and designing them to match our main characters. We searched through different websites and settled on a model through cgtrader.com. Afterward, we used UV mapping through Procreate, a drawing program on the iPad, to customize the model to create two characters: a Princeton student wearing a tiger sweatshirt and Eisgruber with his iconic suit and tie. These models are the heart of our game, being included in every scene, so we wanted their textures to be unique to our game rather than uploading models with pre-mapped textures.

Figure 1, 2: 3D Model of the player from a front view and ¾ view

Figure 3, 4: 3D Model of Eisgruber from a front view and ¾ view

Once the design of the characters was finalized, we created new objects in the code, one for the player and another for Eisgruber. In these object files, we imported the model using GLTFLoader and inserted them into our scenes.

2. Scene Design and Puzzles

For establishing the scenes of the puzzle rooms, we set up an OrthographicCamera using parameters dependent on the user's window size. We set the camera position to be slightly tilted down rather than straight on, making the whole scene look more three-dimensional.

There are six scenes in our final game: Title scene, Introduction scene, Puzzle one, Puzzle two, Puzzle three, and the ending scene. Each of these either adds to the plot of the game or can be interacted with in order to advance. We typically had two formats for these scenes. In the first format, we place our Eisgruber model in front of the character and set the text to progress each time the user presses any key. These scenes were relatively simple, composed of just the model of Eisgruber staring intimidatingly straight at the user, a background color, and text boxes.

Figure 5: Introduction scene

The second format of scenes we used includes both character objects, alphabet blocks, a place for the answer blocks, and a ground plane. We first set up the ground

floor as a vertical plane that is mapped with a repeating concrete texture. We then placed 3D objects in this vertical plane and tilted the camera slightly downwards, this created a similar look to the game A Link Between Worlds and allowed the player to see all boxes at any point in the game. These boxes, labeled with alphabetical letters, are essential to the game's function.

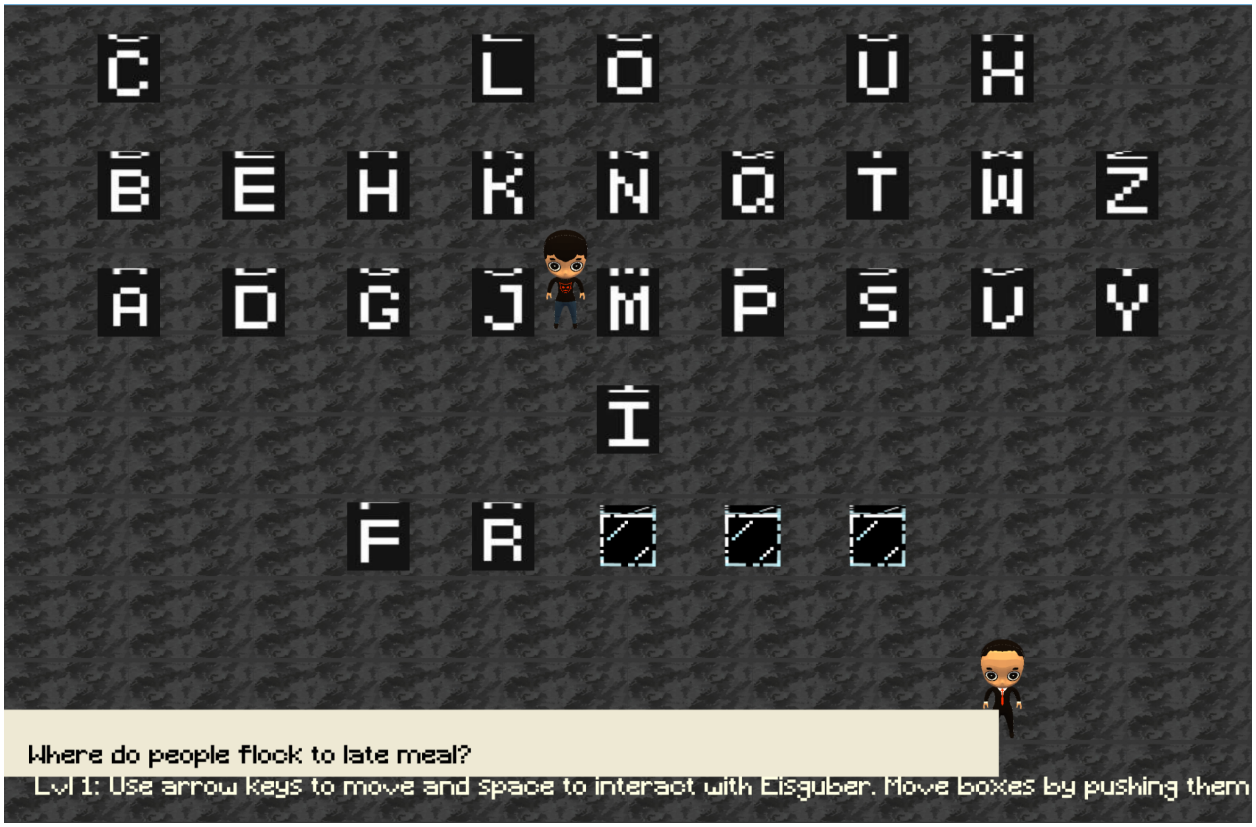

Figure 6: Puzzle One: “Where do people flock to late meal?”

Each puzzle, given by Eisgruber, is solved by pushing the correct sequence of lettered blocks into the correct hole denoted by the glass blocks. To confirm the validity of the answer, we first assess if the box is in a hole by seeing if their bounding boxes intersect. After that, we check if the correct box is matched with the correct hole using their array indices. This changes depending on the puzzle and the answer.

3. Player and Box Movement

We coded the player movement using an onKeyDown event listener for presses of the up, down, left, and right arrow keys. Since the scene is actually a vertical plane, the up and down arrows corresponded to up and down movement rather than forwards and backward like in other 3D games. In the case that a directional key was pressed,

we changed the character's position to one unit in the desired direction and then rotated the model to match the walking direction. Each time the character moved, we also checked for a box collision; if there was a collision, we moved the box's position by one unit in the direction the player was heading. If no collision was detected, the player moves one unit forward. All of these features combined with the layout of the scene gave the illusion of walking through the map when the only movement was up, down, left, and right.

In order to avoid the character from walking off the map or through objects, we also keep track of a bounding box for the player. This box is invisible and moves synchronously with the model assessing if another bounding box has been intercepted or touched. We also gave Eisgruber a matching bounding box, however, Eisgruber remains stationary throughout the game.

4. Collisions

There were a few types of collisions we had to be aware of in our game: player with box, box with box, players and boxes with the scene, and player and Eisgruber. In general, we assigned a bounding box to every object in our scene, including the scene itself to handle scene collisions. In the same onKeyDown event listener that updates the position of the character, we check for the possibility of collisions of all types.

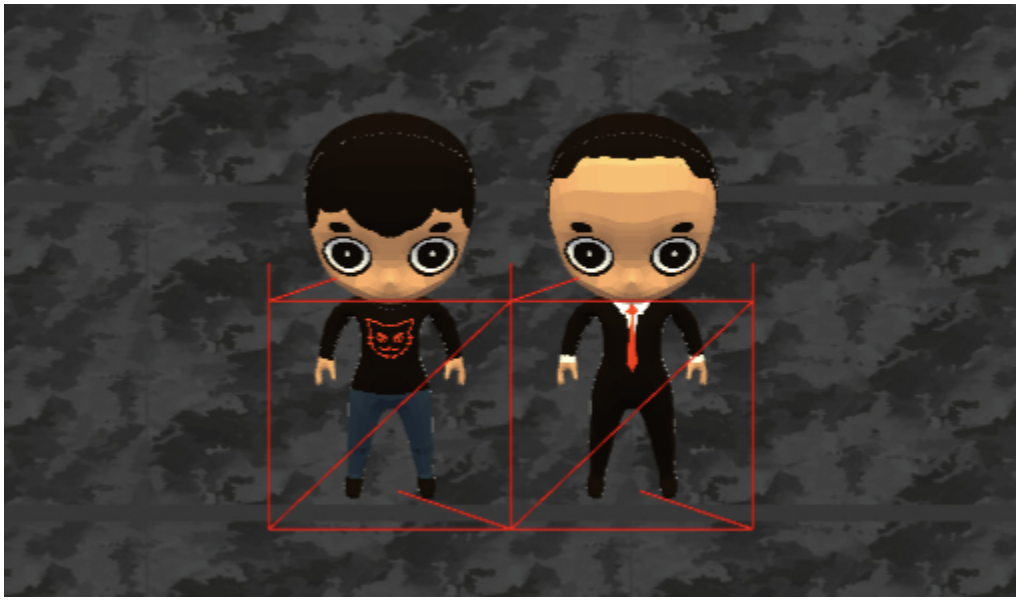

Figure 7: Bounding box of both characters

Player with Box

For collisions, we used bounding boxes to see if there were any intersections between different objects and players. The player had their own bounding box following it with each movement. Therefore, in every direction it moved, so did the bounding box. We tested this by first having a wireframe bounding box follow the user and then gave it transparent properties and setting its opacity to zero so it would not be seen in the final version. We set each box to a whole numbered position, giving the game a grid-like structure, and since the movement of both players and blocks is set to a speed of one unit, our game stays consistent with this grid structure. Since the scene is aligned perfectly with a grid layout, we first had issues with collisions of bounding boxes, as objects would be touching perfectly with the bounding box if it was adjacent to it. We first dealt with this by checking whether they intersect after moving and are in the middle position of the object. For example, if we were going up, we would check if we would intersect the box after we move, if we were between the boundingBox.max.x and boundingBox.min.x, and that we were not already above the box (as then they would not be able to move if they were above the box). We had an array of boxes so we loop through the array to check these cases. If so, we would then block the player from moving by not changing their position or allow the player to push the block. We later discovered a simpler method where we can shrink the bounding boxes so that they are not intersecting when adjacent, resolving many issues. We implemented this solution, later on, to detect collisions on whether the user inputted the correct letters to the right spots. Furthermore, to continuously ensure bounding boxes were updated with movement, we used the addToUpdateList function to add it to the update function. If the object in the update function had a bounding box (in our case are boxes and the player), we then used the applyMarix4 function to keep it updated.

Box with Box and Answer Blocks

To ensure boxes do not merge with each other, we perform another step after checking the collision of the player with the box. If the player did push a box, we save the current box that they push. Then, we go through to check if there are any boxes that are intersecting the saved pushed box now and, if there are, we move the position back to its original place (making it not move at all if the player tries to push a box in a direction where there is another box in place. Furthermore, we also create holes where the player can insert their answer to. For this, we did the simpler method of shrinking the bounding box and checking if the correct corresponding box matches the current answer slot it currently was at. If it is, then the player gets to move on to the next scene after another keyboard input.

Player and Boxes with Scene

For the scene, we wanted the player to stay within the bounds of the game. To make this possible, we create a bounding box for the land. Through this, we can then obtain the land's max and min x and y coordinates. Within player and box movement, we check that they do not go over these max or min coordinates (for example: when the player is moving up, we check that land.boundingBox.max.y is less than player.positionAfterMoving.y). This way, the players and boxes are bound to the scene of the game.

Player with Eisgruber

For the Eisgruber, we used the same logic that blocked the user from the boxes. Therefore, we created a more general "obstacle" array where we would loop through all obstacles and check if there were any elements blocking the player's way, Eisgruber being one of them, so the player could not move forward. If there was an obstacle, the player's position would not be updated.

5. Scene Switching

To implement scene switching, we created a general Scenes.js file to manage all the different scenes we had for our game. The file imports all the different scenes we have, from Title to the puzzle rooms, and stores keys of the scene to a scenes array. Then, using a switchScene(key) function, it can remove the current scene happening currently, promoting culling so it does not have to render all scenes at once. After removing the scene, the function then calls upon the new scene using the key with the user passed-in parameter to access the next needed scene in the previously established scenes array, and then add it to current events. To provide an example with our game, we first start off with "TitleScene" and then, afterward, we call switchScene(SceneOne) to move to the next puzzle screen.

6. Dialogue

For dialogue, we used this mainly in the introduction and conclusion scenes. Through this, we created a textMesh and loaded an initial message. Afterward, we would have an event listener to check when the player inputs any button and, if they do, the dialogue will progress. More specifically, we have a counter after each press that increments and we use this to track what dialogue option the player is currently in and also to remove the previous dialogue or text mesh that was on screen. Once the player reaches the end or up to a certain number of the counter, we then switch to the next scene. We also included a dialogue pop-up when the player interacts (presses

SpaceBar when next to Eisgruber) with Eisgruber during the puzzle so that he would say the riddle through event listeners and bounding box intersections.

7. Audio

In order to generate the appropriate atmosphere, we added music to each scene of the game. To do so, we created a new AudioListener object. Because Chrome required user input prior to playing sound, we started playing the music within onKeyDown event listeners. Most of the scenes use eerie dungeon music which makes the game more suspenseful and eerie. The audio is set to loop so that it is continuously playing throughout the game. At the end of the game, after the player has solved all of the puzzles and is released from the basement of Nassau hall, Old Nassau plays in celebration.

Results

Our resulting game runs smoothly and has graphics consistent with our goal theme. Overall, we met the expectations of our MVP as well as completed some of our reach goals of implementing multiple puzzle rooms. And after allowing a few people to play our game, we added extra instructional text boxes that label characters and describe how to move and interact with objects in the game.

Discussion and Conclusion

Overall we are proud of the game we managed to create, and although we were able to successfully code it, there are a few things we wish we approached differently. While coding the verification of answers, we realized that our box collision calculation would have been made a lot simpler if we had each box be slightly smaller than the unit size. For example, sizing each box to be [0.9, 0.9, 0.9] rather than [1, 1, 1]. Since each scene in our game forms a perfect grid, and each player and box moves on this grid, objects that are next to each other are also inherently intersecting since the bounding boxes were also one unit large. We ended up using this method for the answer calculation; each letter slot, denoted by glass blocks, is of size [0.9, 0.9, 0.9] which means that the answer is only correct if the letter block is moved on top of the glass block. This also meant that once the letter block was placed, the glass block was no longer visible.

As for the future developments of this game, we would like to improve all of our collision calculations to reflect what we discovered in the last section, as well as add more features to the game such as different formats of puzzles, sound effects, more storylines, and more characters.

Contributions

Danica worked with scene transitioning and logic (establishing scenes.js), dialogue boxes, layout, and logic, introduction and ending scene position and layout, event key listeners controls, modularizing code, and adding music.

Lauren worked with coding the player movement, designing and UV mapping the models, title screen graphics, modularizing code, collisions, adding textures, including instructions for users, and adding music.

Warren worked with coding the base code with player movement, creating an array of boxes, the starting scene and camera, collisions between characters, boxes, and scenes, coding the multiple puzzle rooms and creating riddles, and adding music.

Credits

Inspiration:

- Baba is You
- The Legend of Zelda: A Link Between Worlds
- Untitled Goose Game

Code References:

- Three.js Documentation: https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene
- Princetonmon: https://github.com/stephanieyen/princetemon

3D Models:

- https://www.cgtrader.com/free-3d-models/character/child/chibi-body

Audio:

- Dungeon Music: “Welcome to the Mansion” https://soundimage.org/horrorsurreal/
- Old Nassau: https://www.youtube.com/watch?v=eWJmDZv6RIQ