BucketBall Final Writeup

COS 426 Final Project - Emre Cakir and Labib Hussain

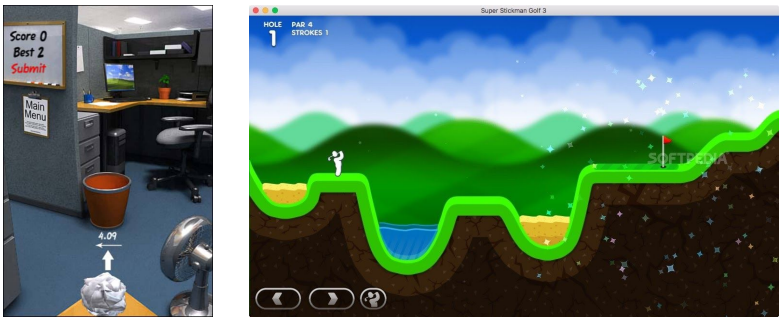
https://cakirmehmete.github.io/BucketBall/

Abstract

BucketBall is a mini-golf video game in which players try to get a golf ball into a hole in the fewest amount of attempts. BucketBall is implemented using the ThreeJS and CannonJS libraries to simulate a world in which the golf ball realistically interacts with its environment, from simple 3D geometries to more complicated polygonal meshes. The players can interact with the video game by using the keyboard to shoot the ball in any given direction and adjust their view using the mouse. BucketBall also consists of levels with their own distinct themes, providing players with a fun video game experience.

Introduction

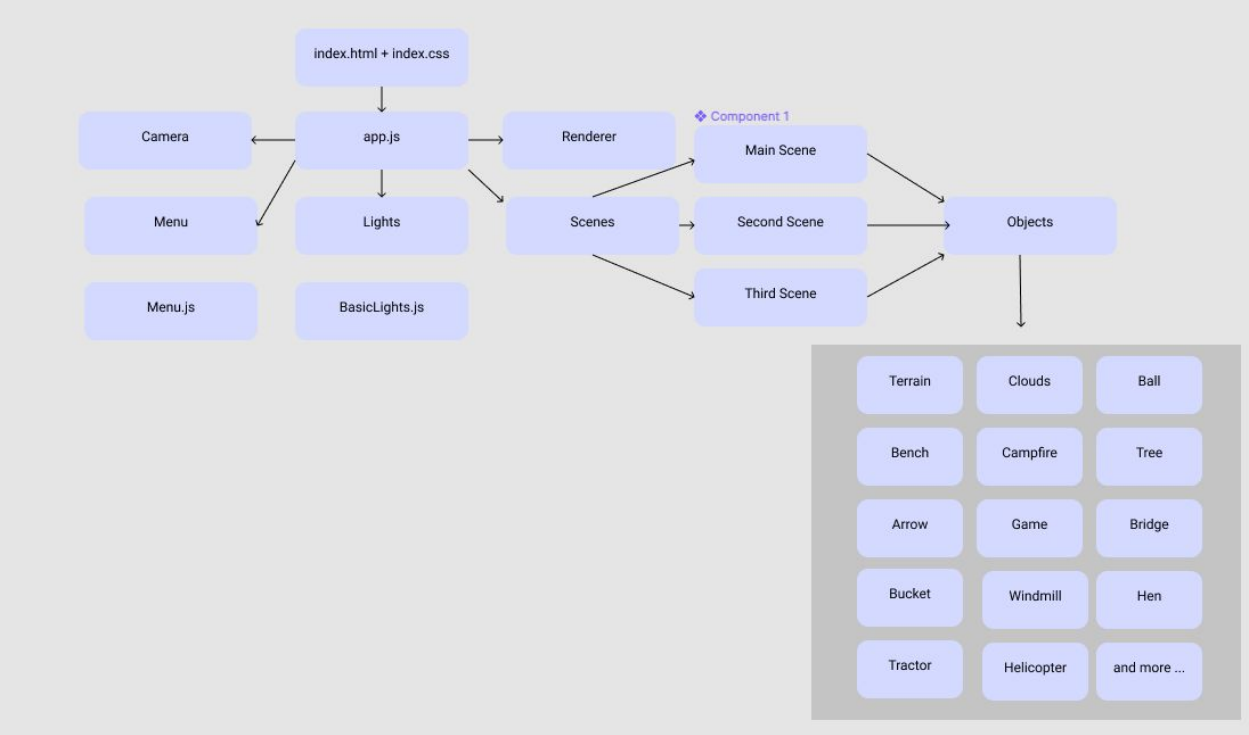
The goal of BucketBall is to provide players with a virtual mini-golf experience by designing levels that provide challenging gameplay and theme-based designs. The inspiration for this project came from games such as the iOS *Super Stickman Golf* and *Paper Toss* games, both of which are pictured in the figure below. However, we wanted to provide a mini-golf video game that focused on various themes as players reach new levels. Games such as *Super Stickman Golf* and *Paper Toss* either provided players with a complete 2D experience or complicated ball control and these were some of the issues we wanted to resolve in BucketBall. The basis of our project was to provide the some of the golf “mechanics” in the aforementioned video games (such as adjusting shot power and angle), but combine the challenging gameplay with themed, 3D level design that the player can experience.



Paper Toss (Left) and Super Stickman Golf (Right)

Our approach to making BucketBall was fairly straightforward: we wanted to develop the game so that players would be able to assess the level design before they made any decisions about how they would shoot the golf ball to get it into the hole. As a result, we wanted the

camera of the video game to be positioned so that the full level was visible, but also provide users the option to change the camera point of view so that they can take have a better view of the field between the golf ball and the hole. We also wanted to provide players with realistic mini-golf experiences, so we included various obstacles that the ball can realistically interact with such as windmills, bridges, etc. As for the core mechanic behind shooting the golf ball, we wanted to provide users with an arrow that visually indicates the path the ball would take when it is shot, but in a way that does not compensate the challenge of winning the level. With this approach, we believe that players who are looking for an online mini-golf game would enjoy playing this game.



A simplified architecture of BucketBall

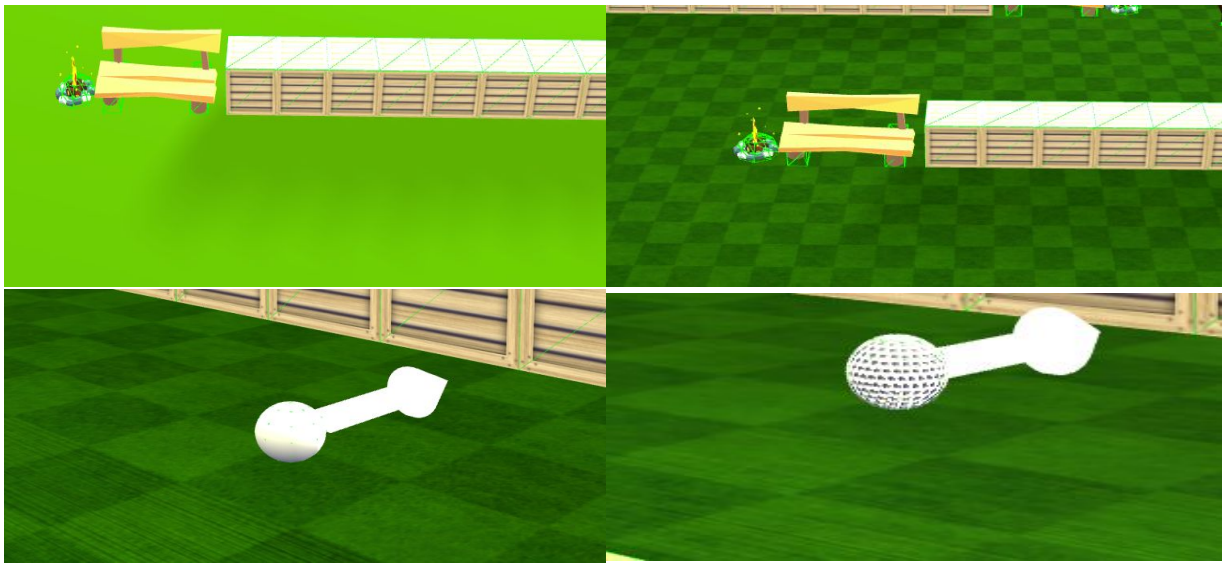
Methodology

Scene Management (Mesh/Lighting and Shadows/Textures/Materials/etc)

Scene management is a fairly broad category within the game, so we will discuss the more important features and how they are implemented. With respect to the basic structure of the different levels/scenes, we had a Terrain object that was created using the plane and box geometries in which all other mesh/objects in our scene would be put on. It is important to note that we oriented the plane to be aligned with the X-Z axis (positive Y facing upwards) which required us to rotate the plane geometry along the X-axis. Implementing most objects within our scenes were pretty straightforward by comprising each mesh with a given geometry and material. Above this section, you will find a picture describing the architecture of our project.

One feature we wanted to incorporate was making the terrain have that traditional, mini-golf felt feel. In order to do this, we added a green-checkerboard like texture to the material of the terrain object. However, to give the terrain a “felt-like” look to it, we used a grass normal map for the normal mapping of the terrain material. With the green-checkerboard texture and the grass normal mapping, we were able to achieve a field that looked much like traditional mini-golf stages. Similarly, a bump map was also used on the ball to make it look like a traditional golf ball. To achieve this golf ball look, only a bump map was used (no texture map or normal map was added).

Another feature we wanted to incorporate was for objects to cast shadows onto the terrain. In order to implement this, we first had to add a light to the scene that was capable of casting shadows. The starter code had initialized a `hemisphereLight` and `ambientLight`, but neither are capable of casting shadows. As a result, we added a Spotlight that would serve the purpose of enabling objects to cast shadows. By enabling the spotlight to cast soft shadows (setting the `shadowMap` of the spotlight to `PCFSoftShadowMap`), setting each mesh’s `castShadow` property to true, and setting the terrain’s `receiveShadow` to true, soft shadows would be projected onto the scene’s terrain. It is important to note that external meshes sourced from Google poly were unable to cast shadows, despite setting their `castShadow` properties to true. This was discussed with a lab TA who suggested we mention this in our writeup.



Terrain/Ball with no texture or normal/bump mapping (left).
Terrain/Ball with texture and normal/bump mapping (right).
Both terrain images have shadows enabled.

Ball Physics / Shooting the ball

One of the original goals of our project was to model the trajectory of a golf ball using projectile motion. We initially approached this using the techniques from Assignment 5. Using the animationHandler at each frame we added a gravity force to the ball. The ball kept track of the net forces acting on it and used it during verlet integration to approximate the position of the ball after applying the forces. We adopted the Assignment 5 code to work with our project and discovered the ball was not following projectile motion. The ball would follow a parabola for the first few time steps, but then it would fall straight down. We tried playing around with the

constants used during verlet integration, but could not get it to work properly. We did some more research and discovered Euler integration.

We found a project online modeling golf ball projectile motion using Euler integration, so we decided to switch to it from verlet integration. We used the Euler integration formulas from Wikipedia and updated the state of the ball to include the current velocity and acceleration. This resulted in a much more realistic trajectory for the ball. After this we added a drag force and spin force to the ball using wikipedia to determine the constants for a golf ball moving through air. This resulted in realistic golf ball motion, however we had yet to code the collisions. After working on coding the collisions on our own, we determined that it would be better to use CannonJS. We tried to use our custom physics for the ball instead of CannonJS' physics, but this ended up being incredibly difficult since decoupling the ball physics from the scene collisions is impossible.

This latest setback meant we needed to switch the ball physics to CannonJS. We created a body that was the same size as the ball mesh and at every frame update the ball mesh to the position of the body.

Shot Indicator - Arrow

We had two versions of an arrow: a trajectory based one and a straight arrow. The trajectory based arrow used our custom physics implementation to draw the actual trajectory of the ball if it were shot at that angle and power. Essentially every time the trajectory was changed the ball would calculate the next 100 positions it would be in if the ball were to be shot. These potential positions were stored and used in the shot indicator. There we created a small sphere mesh at each position to create the breadcrumb look.

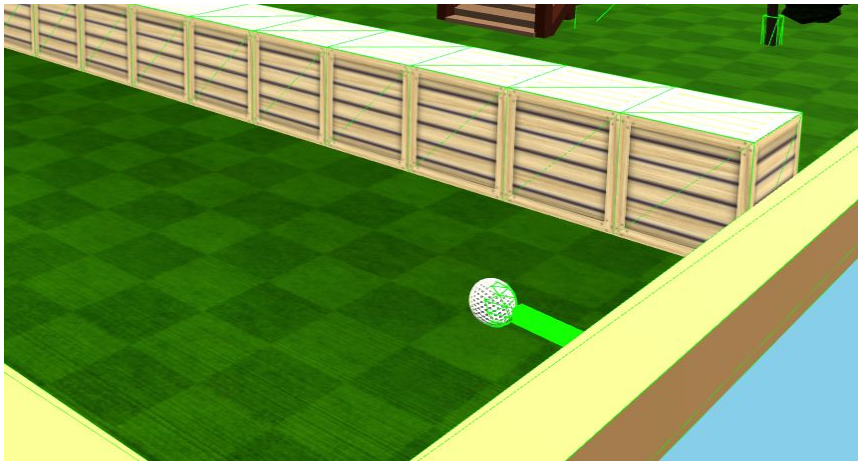
When we switched to the CannonJS physics this approach became much more difficult and we instead opted to create an arrow using ThreeJS geometries and rotating it using a pivot point at the ball.

Ball and Environment Collisions

A core element of our game is collisions, specifically how the ball collides with other meshes in the scene to simulate real-life mini golf. We had a step-by-step approach to handle collisions, and was handled with the help of the CannonJS library. The core element of CannonJS collisions were 'bodies' that we could add to our system and handle collisions for us. However, CannonJS bodies are not visible, and are only position in the scene to indicate where collision would occur. As a result, whenever we added a collidable ThreeJS mesh to our scene, we had to make sure that we initialized an associated CannonJS body with it. Once we initialized the mesh object and its corresponding CannonJS body, threeJS would be responsible for rendering the mesh and cannonJs would be responsible for handling the collisions between objects.

The aforementioned approach was applied to a vast amount of objects across all three of our levels/scenes, but we will use the golf ball and crate objects of our system to explain how it was implemented. Before we could add CannonJS bodies to the ball or crates, we first made a call to our `setupCannon()` function that initialized a CannonJS "world" that will keep track of all our physical/collidable objects and a few fields responsible for setting the gravity of the system. Then in our `setupBall()` and `setupCrates()` functions, we first used ThreeJS to add a mesh to the scene. Once the position of our ball and crate mesh were set, we use CannonJS to create a corresponding body at the same position of our mesh object. In this case, we used a box-shaped body for the crate and a sphere-shaped body for the ball. We then added these bodies to the physical world that was initialized in `setupCannon()`. The position

of the invisible CannonJS bodies are outlined in green wireframes in the picture below. It is important to note that collisions were used to also create more complex interactions between the ball and the scene. The bridge from Level 2 is an example of using primitive CannonJS bodies and putting them together to create a collidable geometry that allows the ball to go up and down the steps of the bridge.



Green wireframes outline the position of CannonJS bodies.

However, we had to add an extra step to meshes that moved, particularly our golf ball. In order to ensure the mesh's position would update whenever a force was applied to the CannonJS body, we copied the position and orientation of the body at each time step and set it to the position and orientation of the mesh. In this sense, we're applying the physics of the ball to the body itself, and ThreeJS is just rendering the ball for us.

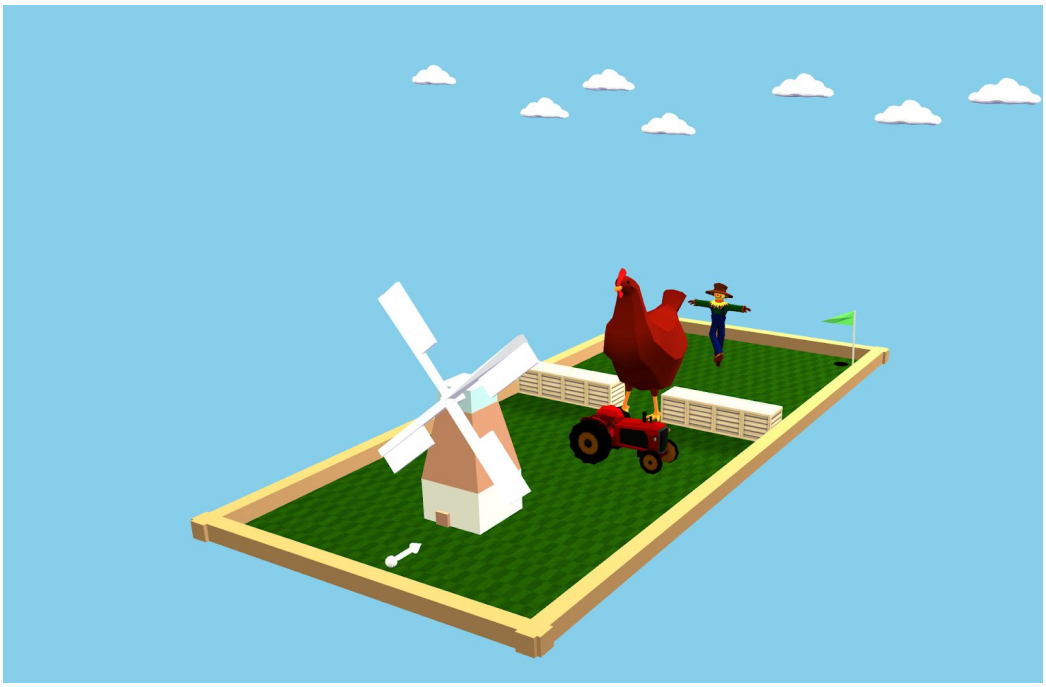
Animating External Meshes with Blender

Several of the meshes that we used contained more elements than we wanted to include in the game so removed them using Blender. After importing the gltf files into Blender, we selected all the faces we wanted removed and deleted them. Then we could export the mesh as a glb file and use it in the game.

We animated the windmill mesh using the Blender Animation pane. We separated the base of the windmill from the turbine and then created a pivot point in the center of the turbine. Then we added two keyframes to the z rotation of the turbine, one at time 0 at angle 0 and one at time 60 with angle 360. Then we changed the extrapolation of the animation to be linear to the turbine spun realistically. Finally we exported the glb file with the animations included and used the created a new animation using the gltfLoader.

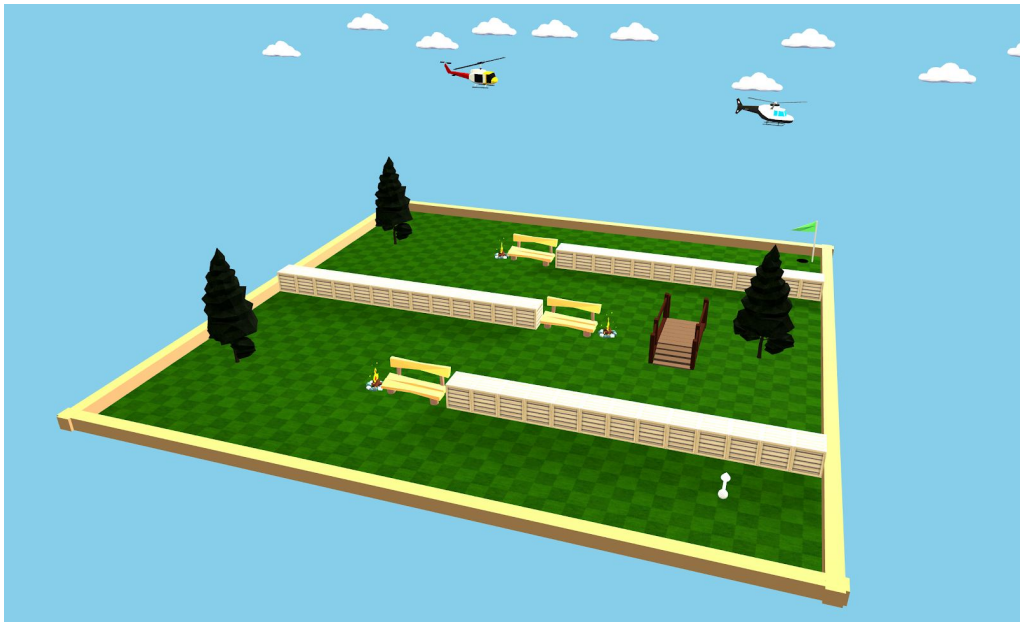
Level Designs and Themes

We wanted each level to have its own theme. We used a farm theme in the first level and included a windmill, tractor, rooster, and scarecrow. The meshes for these objects were found using Google Poly. We used an online gltf to glb converter so deployment would be easier. We animated the windmill using Blender. We animated the tractor using the animationHandler and moving its z position at each timestep. For each mesh we also created CannonJS bodies to match the size and position of each mesh. We also updated the position of the tractor body when the mesh's position was updated.



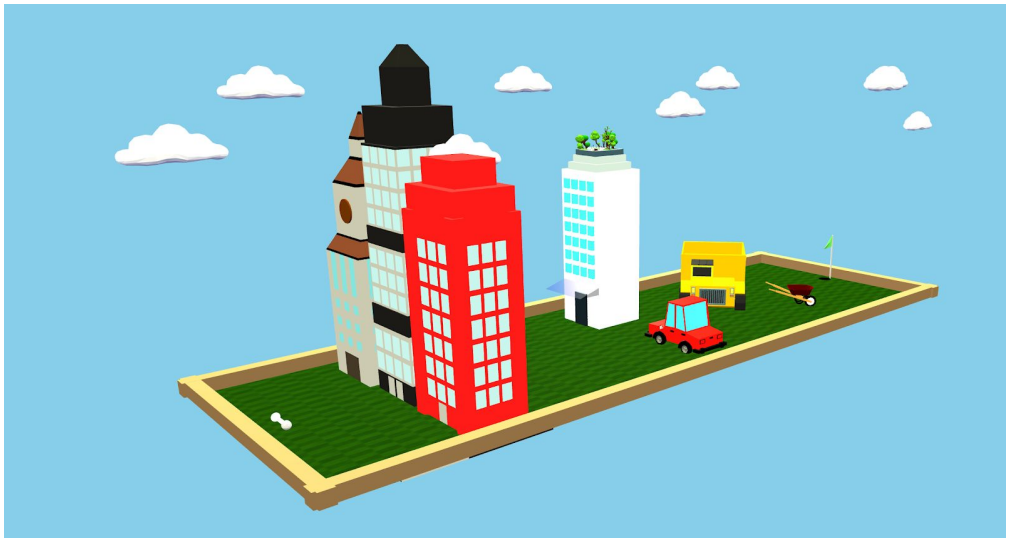
Level 1

Our second level has an outdoor theme and we included benches, trees, campfires, and a bridge from Google Poly. The design process was very similar to the first level, however this level includes multiple ways to get to the hole. The player can take the bride or go underneath the benches as a shortcut.



Our third level has a city theme and includes skyscrapers, a construction truck, a wheelbarrow, and a car from Google Poly. Once again this level was created using the same process as the

last two levels. We did want to make this level extra challenging though so we made the distance between the skyscrapers as small as possible.



Level 3

Gameplay and in-game Text

The gameplay of BucketBall consisted of checking for win conditions, updating in-game text, and switching between different levels. Most game-related elements, such as the `updateAttempt()`, `resetGameText()`, `checkWinCondition(ball, bucket)` were located in a Game class that could be imported in the various scenes and used to handle the gameplay components of the scene.

In order to check the win condition, which was the checking for when the ball falls into the hole, we had to use ThreeJS to determine the positions of the ball and hole. Since the field of view of the game is oriented in the X-Z axis, we had to extract the x and z positions of the ball and the hole in the current time step. In order to determine if the ball "falls" into the hole, we check to see if the x-z position of the ball is within the radius of the hole's x-z position. This was made possible using the distance formula: for each time step in the scene, the `checkWinCondition(ball, bucket)` calculates the distance between the ball and the hole on X-Z axis, and then checks the distance against the hole's radius. If the distance of the ball is less than the hole's radius, the ball was considered to be in the hole. If this check ever fulfilled, we set the visible property of the ball's mesh to false to simulate it falling into the hole, and update the underlying HTML accordingly to display a message that displays a win and how many attempts were made.

Naturally, the player also needs a way to keep track of how many attempts they made. We added a div element to the underlying HTML of the game that displays the attempts made. In order to update the number of attempts, a function `updateAttempt()` updates the attempt number within the Game class' state, and displays the current attempt number by selecting the appropriate div element whenever the user releases their spacebar.

Most in-game text was implemented using the approach described with the attempt counter in the previous paragraph. In addition to using underlying HTML to display game text, The menu screen at the start of the game also includes a `startGame()` function that is attached to the play button. Whenever the play button is clicked, this function sets the visibility of the menu div to false and renders the first level's scene.

As for switching between levels, we use the `updateLevel()` function in `app.js` and pass it as an argument to each scene dedicated to a specific level. Whenever the winning state of the game is triggered, this `updateLevel()` increments a list of scenes to determine the next level (or its dedicated scene, rather) to display. It is important to note that before displaying the next scene, we make a dispose call to the current scene to prevent any memory leaks that may occur because of previous references to ThreeJS geometries and materials.

Power Slider Animation with Dat.gui

One of the core elements of shooting the golf ball is the adjusting the power that is applied to its shot. For example, if the player were very far away from the hole, they could use a larger power to cover greater distances, but if the player were very close to the hole, they can fine-tune the power to give it a small force. As such, we wanted to allow the players to adjust the power that is applied to the ball by holding down the space bar, and releasing the space bar when the slider is at their desired power.

To implement the power slider, we decided to use the Dat.gui element. Using Dat.gui as opposed to an HTML element was purely stylistic, as we felt the Dat.gui element had the aesthetic we wanted. Whenever the spacebar is pressed down, we have an event handler that sets the `spacebardown` state within the scene to true. We then implemented a function `animatePowerSlider(timeStamp)` that oscillates the `power` value within the state using a sin function. We had to fine tune the sine function by applying a transform that keeps the power range between 1 and 10, (which was done using a phase shift of 5 and frequency of 5). The sine function took the timeStamp of the scene as input, and we divided the timestamp by a magnitude so that the slider would oscillate between the minimum and maximum values of 1 and 10 smoothly (not doing this update would animate the power slider too quickly).

This `animatePowerSlider(timeStamp)` was then called in the scene's `update(timeStamp)` function, but only whenever the spacebar was held down (which was done through a check to the `spacebardown` state).

Future Work and Features

Much of the core functionality that we intended to add to the game was implemented, however we feel that there could be some additions in the future that could improve the game's experience. Here is a list of some ideas for future work:

- Making the game a side by side multiplayer in which both players try to complete each level in the fewest amount of attempts. Player with fewer attempts wins
- Adding more complicated terrains like water or fescue that realistically depicts the buoyancy or weight of the ball
- Changing the position of the hole in real-time

Results and Discussion

In effort to gather feedback on our project, we sent out a deployed version to our friends and family and asked them to report their experiences when playing the game. We measured success on two factors: the difficulty of the level design and how realistic the interactions

between the golf ball and its environment were. Much of the feedback indicated that players completed the given levels between 10 to 20 attempts, which was the range of attempts we were striving towards. This means that the game was relatively challenging, but still doable for most players. The feedback also suggested that the interactions between the ball and the environment kept players engaged. One player mentioned that the using the bridge to get the ball through Level 2 was something they enjoyed, however they did mention that they did not know the bridge was capable of being used in such a way. This feedback suggests that the collisions between the ball and different parts of the environment were something that kept the user interested, but that we should have done a better job in our level design so that these interactions were more apparent.

Overall, the approach we took helped us produce a realistic, virtual mini-golf game. We were able to incorporate collisions and physics into the game that enabled the player to complete levels in various ways. However, we do feel that the project could have been improved if we had added the option for the user to change between different types of balls or created a dynamic terrain in which certain parts have more friction than others. These additions were stretch goals that we originally planned to incorporate into the game, but due to the time that was spent figuring out the 3D physics of the ball, we did not have enough time to implement them.

Throughout this project we learned how to divide a large project into different parts so that we could both work simultaneously on the project. This was crucial for getting this project finished since we did not have the time to wait for each other to finish a section before the other could start. We also learned a lot about deploying a project to a website. We ran into a lot of issues with hard coding paths and missing textures. Lastly, we learned a lot about creating aesthetically-pleasing scenes and the importance of finding colors that work well together. This project challenged us to maintain a strict low poly look and it really helped develop our design self.

Contributions

Both members of the BucketBall team divide the work evenly among themselves. This means that we often found ourselves working on the same features in order to help each other debug. For example, both of us had worked on level-design and game mechanics. However, each of us were responsible for broader categories of the game. Here is a non-exhaustive list of things each of us were in charge of.

Labib -

- Gameplay and In-Game Texts
- Collisions
- Scene Management (Lighting/Shadows/Textures/Materials/Meshes)

Emre -

- Ball Physics
- State Management

- Animations

Works Cited

- ThreeJS - https://threejs.org/
- CannonJS - https://schteppe.github.io/cannon.js/
- Dat.GUI for animating a power slider - https://workshop.chromeexperiments.com/examples/gui/#9--Updating-the-Display-Automatically
- Google Poly Library - https://poly.google.com/
    - https://poly.google.com/view/aJouyp2PTtZ
    - https://poly.google.com/view/5TGoA5N14c5
    - https://poly.google.com/view/8Unya0rw9tR
    - https://poly.google.com/view/7qFs_DjjuVp
    - https://poly.google.com/view/8ywGTrLRgBE
    - https://poly.google.com/view/6DJeZaACYpm
    - https://poly.google.com/view/fbDxapxkwY9
    - https://poly.google.com/view/0fE2T9eRA9V
    - https://poly.google.com/view/5UU85o2PgY6
    - https://poly.google.com/view/dxxHpVXHLZg
    - https://poly.google.com/view/ahT2_ZxtLIq
    - https://poly.google.com/view/cTzINMr0WdS
    - https://poly.google.com/view/eb7b31pjGtQ
    - https://poly.google.com/view/44cGXp6_8WD
    -
- Manual Golf Shot Simulation (for our initial implementation of the ball physics, which is no longer used) - https://github.com/jcole/golf-shot-simulation/tree/master/src