COS 426 Final Project Writeup

Avatar: The Last Infinite Runner

Catherine Yu, Jingwen Guo, Lily Zhang

cyu6.github.io/avatar-runner
https://github.com/cyu6/avatar-runner

Abstract

In this infinite runner game, play as Aang the Last Airbender and overcome obstacles using four different elemental attacks. Control the Avatar using your keyboard. The goal of the game is to survive for as long as possible.

Introduction

Goal

Our goal for this project was to design an infinite runner game based on the show, *Avatar: The Last Airbender*. In this show, characters are able to bend the four elements of water, fire, earth, and air. Thus, we wanted to create a game that would adhere to this theme and require players to bend the elements in order to overcome obstacles. Our goal was to create a fun and engaging game with relatively realistic representations of the elements. In the spirit of *Avatar: The Last Airbender* coming to Netflix, we hope this game encourages others to check out the show!

Previous Work

An infinite runner game is essentially a game that runs forever and only stops when the player performs a certain action (typically colliding with an object). There are many previous iterations of this game format. Perhaps the most popular and well-known infinite runner games are Temple Run, Flappy Bird, and Subway Surfer. The Aviator game on the THREE.js website is also another example of an infinite runner game. Of these approaches, Temple Run is the most realistic, and it additionally provides increasingly difficult game play as time goes on. Flappy Bird and Subway Surfer are both rendered in a more cartoonish style and the difficulty of these games generally doesn't increase as the game goes on. Each of these games focus on the left-right-up-down movements of the player relative to the environment, but don't necessarily ask for any more creative thought. Thus, for our project we challenge the stereotype of infinite runner games by creating a game that doesn't focus on the left-right-up-down movements of the player, but instead asks for more creative problem-solving.

Approach

Similar to most of the other projects, our approach relied on making heavy use of the THREE.js library for creating objects and rendering the scene. This choice was mostly due to the extensive documentation of the library and our familiarity with it through previous assignments in this course. Our high-level approach was to draw inspiration from the animated series in creating an interesting twist on an infinite runner game. Thus, we designed pairs of obstacles and elements to counter each other (fire and water, earth and air). Our approach was to start from the basics of an infinite runner game and add visual elements, and additional game components one by one, since the infinite runner format allows natural extensibility. To create our avatar character with animations corresponding to the various element-bending moves, we used a mix of different applications, including Adobe Fuse, Blender, and Adobe Mixamo. To create the various obstacles, elements, and background scenery components, we relied on a mix of THREE.js primitives, a particle engine, and existing 3D meshes. Overall, our approach was to create a relatively realistic-looking take on the animated series. Due to the tradeoffs between realism and performance, our current product can be laggy on certain computers. During the development process, we did not experience performance issues significant enough to adversely affect gameplay, but it seems on older computers and graphics cards, the game can lag enough to make the game unplayable for some. For example, one person who tested the game on an i3 core could not see the obstacles or the avatar. Thus, our approach to creating a realistic-looking scene works well when the player has a fast computer, since many computationally expensive components are added in quick succession.

Methodology

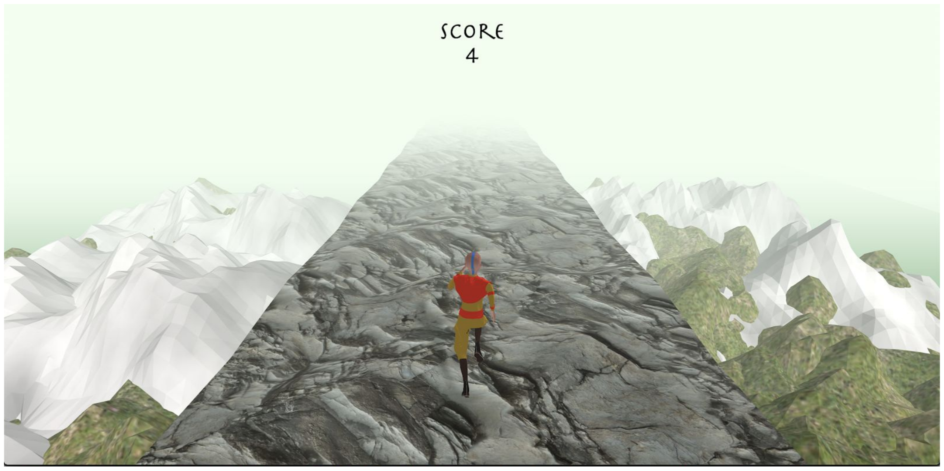
Scene

To set up the scene, we kept most of the starter code. We also based our code off of the scene setup of an infinite runner game tutorial since we wanted to have a similar point of view for the camera and the lights. We ended up disabling OrbitControls so that the camera would be fixed at the runner. We included fog to add to the overall atmosphere and feeling of being high up in the mountains. Lastly, we set a directional light on the Avatar runner, and we included a hemisphere light for overall lighting.

Avatar

Since our game takes direct inspiration from the animated series, *Avatar: the Last Airbender*, we wanted the avatar model of Aang to resemble the original as closely as possible. Our first approach was to search online for existing 3D meshes of the character. However, the ones we found were either not downloadable or did not contain the entire character (e.g. just a model of the head). Thus, we had three options: use a first-person perspective for the game, in which we view the scene through the eyes of the character but cannot see the character himself, create a simplified version of the character using THREE.js primitives, or build our own mesh using software like Blender and Maya. The first approach is the simplest to implement, but we felt that the game would be missing a major link to the source material. The second approach is perhaps a good option for building a very simple character or a highly simplified version of Aang (e.g. 8-bit style), but we felt that it would not fit the overall look we envisioned for the game, as the scenery and elements are more realistic. Thus, we chose the third option, which, while time-intensive, gives us the most control over a key component of the game.

In order to create the character, we researched different existing products and software options. To save time, we created a base character in Adobe Fuse. To add clothes, we imported it into Blender and added clothes using the Modeling Cloth add-on, which implements a cloth simulation for collisions between cloth constructed from primitives and objects set as colliders. We constructed clothes using subdivided planes, sewed them together using the add-on, and set the base character as a collider to implement the cloth simulation. Afterwards we added textures and colors using UV mapping. To complete the look, we added the blue arrow tattoos on Aang's head and hands using a stencil painter. We wanted the character to have specific movements to correspond to the running motion, as well as each type of element-bending. To do this, we imported the character into Adobe Mixamo, a free online tool that auto-rigs humanoid characters and has a large animation library. We downloaded the rigged character and animations, imported them back into blender, and combined them to export as a single GLTF file with multiple animations. Overall, while the process was time intensive, since we had to research different tools and familiarize ourselves with Blender, our final avatar has the look and movements we envisioned in the design process.



Aang running through the mountains

Scenery

We initially looked into procedural generation of terrain and terrain meshes for the scenery on the sides of the ground, but decided against this implementation because the level of detail was unnecessary in the scope of the project. Instead, we found a mountain object and added it along the sides of the ground to provide for the scenery instead, which worked well and did not require more complex structures or implementations. We included a degree of randomness in each mountain's position, rotation, and texture--we included a snow texture for "snowy" mountains.

Keyboard Controls

To detect keyboard controls, we added a keydown listener to our game scene. The Avatar object has a left and right boolean stored in its state. Upon clicking the left or right arrow, the state is updated in the game scene to say true and then the movement is registered within the Avatar's update function. To make the movement from side to side smooth, we created Tween objects that make the Avatar move to the left or right over the span of 1 second. Initially, our approach was just to naively set the avatar's position slightly to the left or right over the span of a few calls to the update function. However, this resulted in choppy movement that did not look good. Thus, we used Tween to make the movement smooth. In our game, the avatar can move from side to side but not up or down, as those movements are not necessary.

Other keys we listen for are the AWEF keys. Each of these keys correspond to an element attack: A for Air, W for Water, E for Earth and F for fire. Upon click of any of these keys, the corresponding element function is called, which initiates the element bending.

Lastly, we also added a keydown listener to the window to detect for space bar clicks. Upon clicking the space bar, the pause function is called which pauses the game.

Gameplay

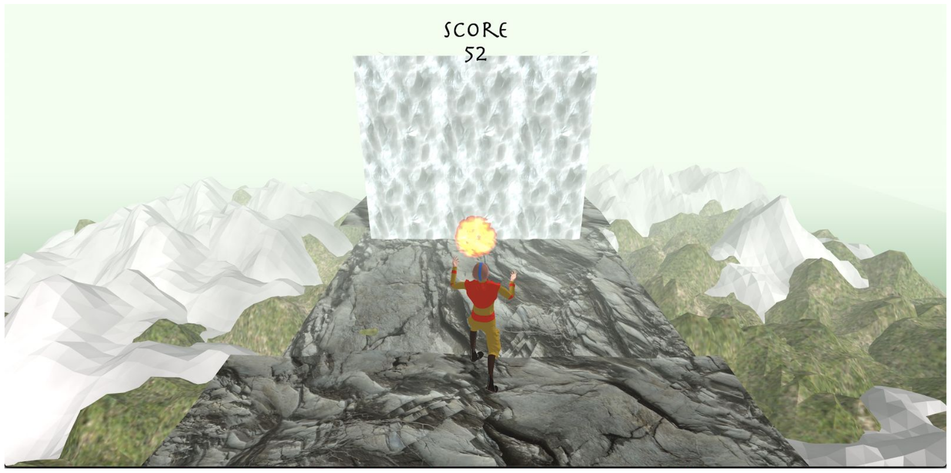
We kept a global variable for the status of the game to check when the game was "ready", "playing", "pause", and "end". When the game was "ready" at the very beginning on the landing page, the avatar would run but no obstacles would be spawned. Once the "let's run" button is clicked on, the game is "playing", so all of the components are being updated, obstacles are being spawned, and the scorekeeper starts running. If the player pauses the game in the middle, everything stops until they unpause the game. When the player "dies" by colliding with an obstacle, the game is in "end" status, and nothing happens until the player clicks on the "play again" button, at which point the game is reset to "playing" status and the other parts of the game are reset, including the avatar, ground, and scene.

Ground

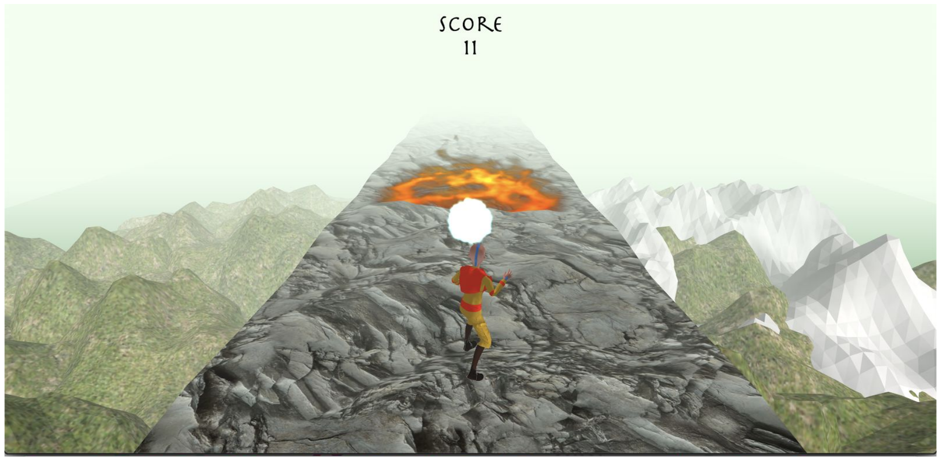
Initially, the ground for the game or "infinite plane" was just composed of a very, very long plane whose position on the z axis would increase slowly so that it looked like the scene was moving. This was a simple approach that would have worked as it would be very unlikely for the game to run long enough to reach the end of the plane. However, during development, we decided we wanted a gap in our plane for our earthbending attack. Thus, instead of this approach, we changed it so that each plane is short. Upon reaching a certain part of the current plane, another plane is created behind the current plane. This makes it seem like there is an infinite plane with numerous gaps in it. This approach is a bit harder to implement, as we need to keep track of which plane is the current plane and also get rid of planes as they pass. However, it allows us to actually simulate an infinite plane that will never end.

Firebending and Waterbending

For firebending and waterbending, we initially envisioned a realistic fireball and a large wave of water, but they both turned out to be more difficult to implement than expected. We first attempted using an animated spritesheet texture on a sphere for a “water ball” and “fireball”, but it was not very realistic. We also searched for fireball objects or meshes, but there were very limited options. Instead, we were able to find a particle engine that used a Points object to simulate special effects, including fireballs. We imported the file but had to convert it from an older version of THREE.js to the newer version used in our project. We found that most shaders and models for realistic water were good for large bodies of water like oceans and not a ball of water, so we decided to use the same engine to generate a water ball, which does not look very realistic but was still a better option than the other two implementations.



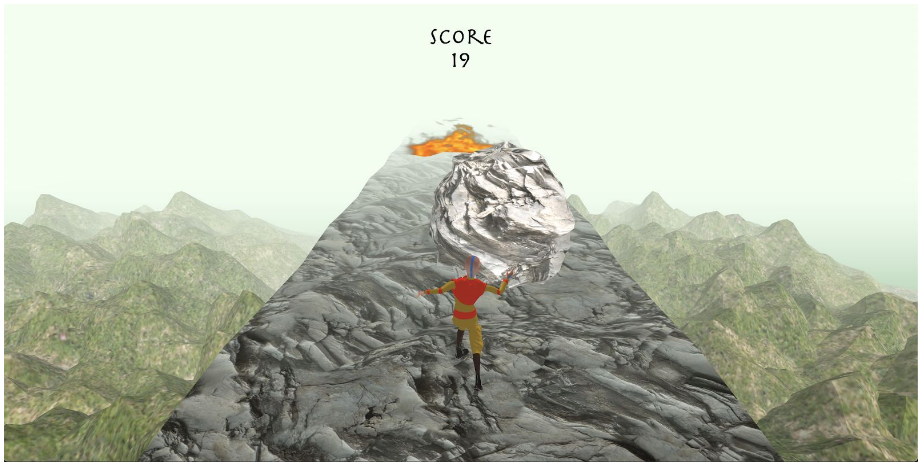
Aang firebending against the ice block



Aang waterbending against fire

Airbending

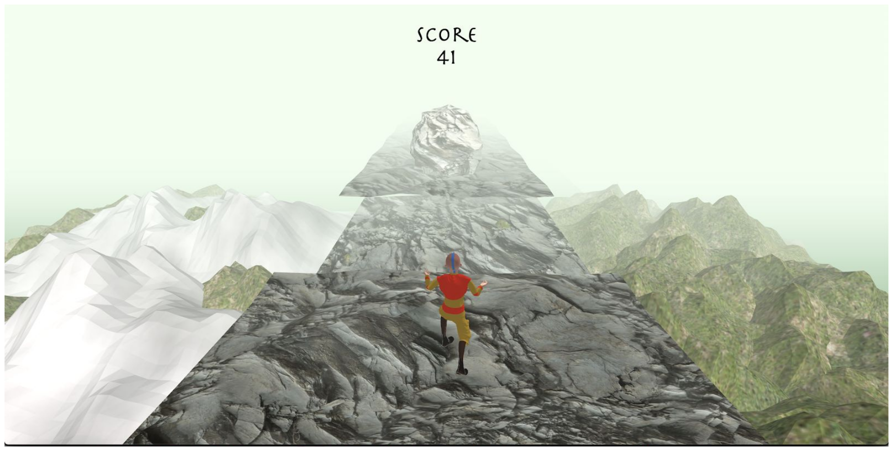
For airbending, we considered a couple of implementations. In the original game design we presented, we planned on representing this element using a jump over chasms in the ground. However, from the presentation feedback, some people thought that would be too simple and recommended making airbending look as involved as the other elements. Thus, we decided to use airbending to create gusts of wind to move rock obstacles out of the path. In our initial implementation, we did not create an object to represent air and iterated through the obstacles until the first rock, on which we performed airbending. However, we quickly realized that this could result in airbending rocks that were very far away, which did not make sense in terms of gameplay. Thus, our current implementation uses a transparent SphereBufferGeometry mesh similar to the fire/waterballs. When we detect a collision with an obstacle, we airbend it if it is a rock. Depending on the current avatar x-coordinate, we airbend the rock to the left or right (opposite side of the avatar) to make the motion more visually interesting.



Aang airbending against rock

Earthbending

For earthbending, we create a bridge between gaps in the plane. We explored a couple of feasible implementations. We considered using a pre-existing mesh found online, but none fit the look of our game exactly, and loading large meshes would contribute to performance issues. We considered an approach of assembling multiple rocks that fly in from different directions to form a bridge, but coordinating their movement and making the joint behavior look physically realistic was challenging. Our current implementation using a single THREE.js geometry with texture and normal mapping. To create the levitation effect in contrast to a stable ground, we used the TweenJS library to mimic the up-and-down motion. We also added a slight back-and-forth bobbing using rotation in the z direction. To simulate the bridge rising from the scene far below our ground, we rapidly increase the y coordinate of the bridge and cap it at the height of the ground. An additional consideration was whether to make the bridge automatically fit the next gap, or to allow the bridge to be formed at a fixed distance from the current avatar position. We decided the second would allow more interesting gameplay, as it would be possible for the play to press the earthbending key too early or too late, which would result in an unideal placement of the bridge and allow the avatar to fall through the gap. Thus, timing of keypresses becomes an important factor in the game.



Aang earthbending to cross the gap

Fire

Multiple approaches were considered for fire. We looked online for particle engines, shaders, and other open source fire implementations. Initially, we tried to implement a VolumetricFire library as linked below. The benefit of this implementation was that the fire's size could be expanded as needed. However, we ended up having a lot of trouble importing this code, and it eventually did not work out. We then moved on to the current approach, a "ray tracing based real-time procedural volumetric fire object", which is actually just represented as a BoxGeometry object. This code was a lot simpler and was also easy to import, and we were able to get a fire on the screen quickly. This fire, however, is really expensive to render and causes lag if it is made larger. Thus, we had to decrease the size of the fire so that additional lag could be avoided.

Ice

Our ice obstacle is a simple BoxBufferGeometry mesh with Phong material, and we tried many different ways to make it look more realistic. We tested applying refraction to the wall based on the THREE.js refraction example, but found it difficult to generate a CubeTexture (which was required for the refraction quality) for the environment because it is constantly moving. We ended up applying a texture map, a displacement map, a normal map, and a specular map to the wall, as well as making it slightly transparent to help make the obstacle look more like ice.

Rock

To create the rock, we considered either using an existing 3D mesh or to create one from ThreeJS primitives. Many of the same tradeoffs existed here as for the bridge used in earthbending. In the end, we decided that better performance from using a primitive was more important, so we used a DodecahedronGeometry as the base, and applied texture, normal, and displacement mapping to distort the shape into a more irregularly shaped rock.

Gap

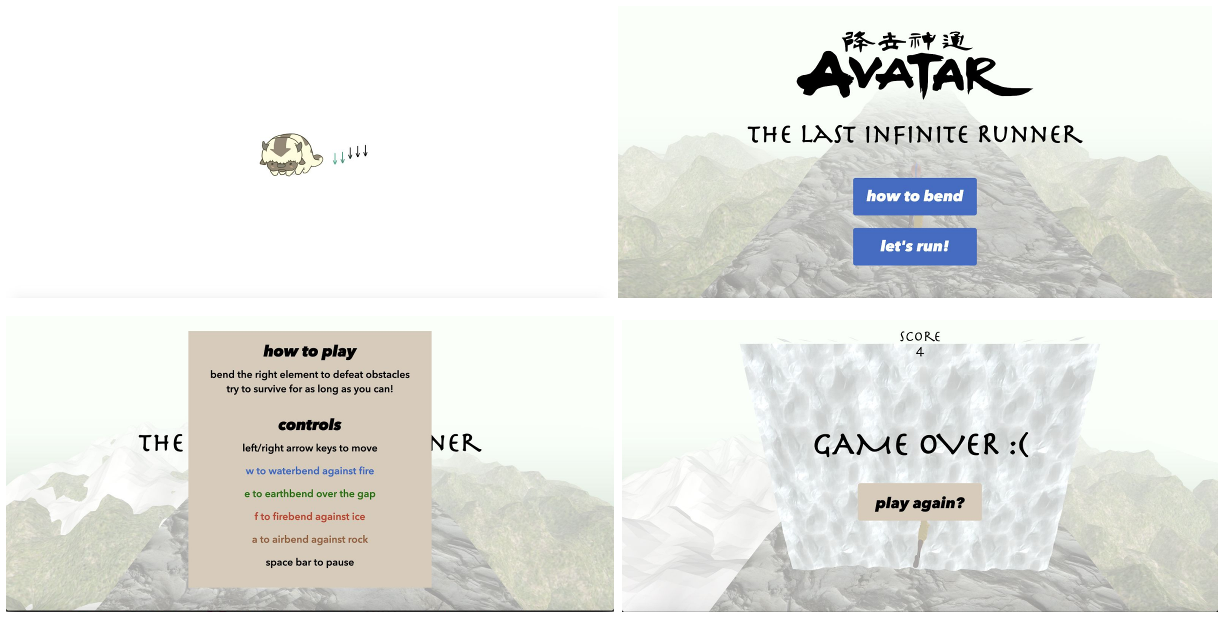
To detect the gap between the planes, we thought about 2 different approaches. In one of them, we would compare the ground's edge position with the avatar's position; if the avatar successfully used earthbending to cross the gap, they would pass safely, otherwise the game would end. Although this approach was simpler in terms of detecting the gap, after implementing the gap detection in the Scene file, we found it was more difficult to integrate the earthbending piece into the logic, as we would have to pass an Earthbending object to the Scene and then check its position against the ground and avatar's positions. Thus, we went with the second approach, in which we simply create an obstacle object for the gap like the other obstacles--a transparent plane placed at the edge of each piece of the ground, and we check it with the Earthbending object with collision detection like the other obstacles. This is slower for our game but works well with Earthbending.

Collisions

We included collision detection between the avatar and all of the obstacles, as well as between each of the specific element-obstacle pairs. A list of obstacles is kept in the Ground object, which is updated when the scene updates and is passed into the avatar and element objects to check for collisions. We initially tried to implement collision detection using ray tracing. However, this didn't work very well as it would detect obstacles too early, and it was also very inefficient to traverse all of the avatar's vertices and send a ray out from each one. Instead, we used bounding boxes of the objects to detect collisions, which was much more reliable and efficient. For the avatar, we go through each of the obstacles and check if the bounding boxes of the avatar and the obstacle intersect--if they do, the game ends. For each of the bending elements, we must check if the obstacle it collided with is the correct corresponding obstacle (i.e. firebending vs. ice, waterbending vs. fire, earthbending vs. gap, airbending vs. rock). If the obstacle is correct, we remove the obstacle from the list of obstacles and we remove the element as well (except for earthbending), otherwise nothing happens.

HTML/CSS

We decided to add a loading screen to our game, so that the elements of the game have enough time to load. Without the loading screen, it is possible that the user might start the game before all the resources have loaded in. Thus, by adding this, we make sure that that can't happen. The current loading screen was adapted from HTML and CSS taken from an online loading screen resource. To add more custom flair, we added a picture of Appa, a character from Avatar: The Last Airbender to the loading animation. For music, we decided against using the THREE.js Audio class. We attempted to add background music with this approach at first, but found that it introduced a lot of lag and also did not start immediately upon calling play. Thus, instead we use standard HTML to add the audio to the game. Once the game is started, we retrieve this element from the HTML and call play so that the music is played. We didn't end up implementing sound effects for the character movements due to time constraints. However, that is something we are looking into adding in the future.
We used the same font as in the TV series for the title font and we used a sans-serif font for the headings and body font. We kept a score counter on the top of the page during the game which was the time passed in seconds for simplicity. Lastly, we had a game over page with a button to play again.



how to play

bend the right element to defeat obstacles
try to survive for as long as you can!

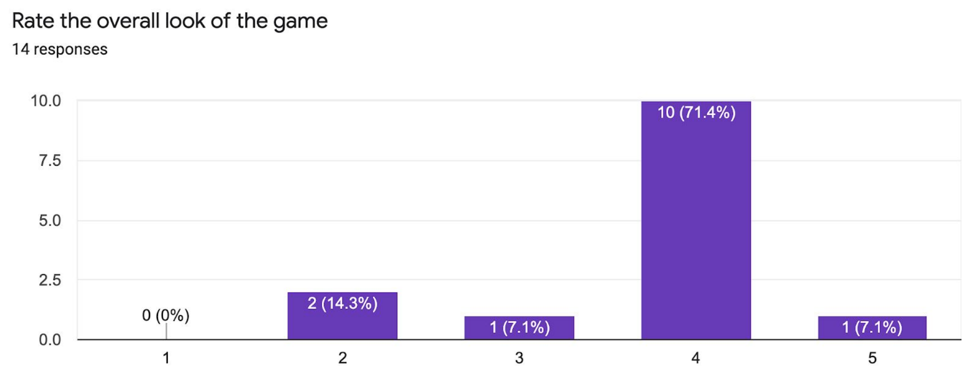
controls

left/right arrow keys to move
w to waterbend against fire
e to earthbend over the gap
f to firebend against ice
a to airbend against rock
space bar to pause

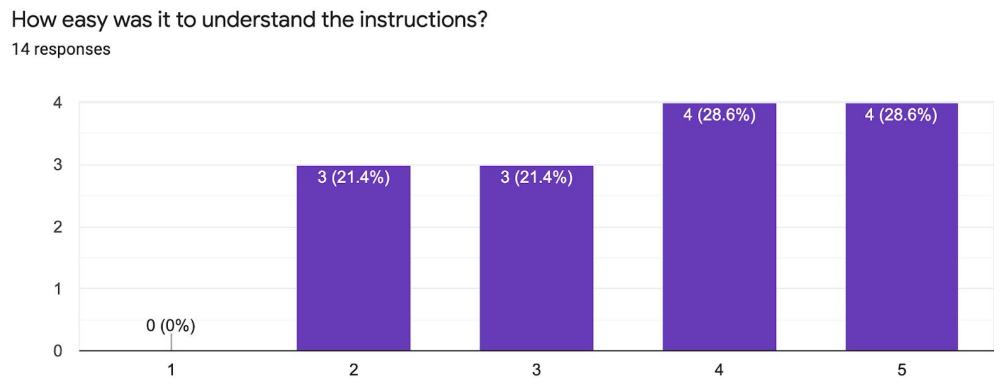
Clockwise from top left: loading page, landing page, instructions, and game over screen

Results

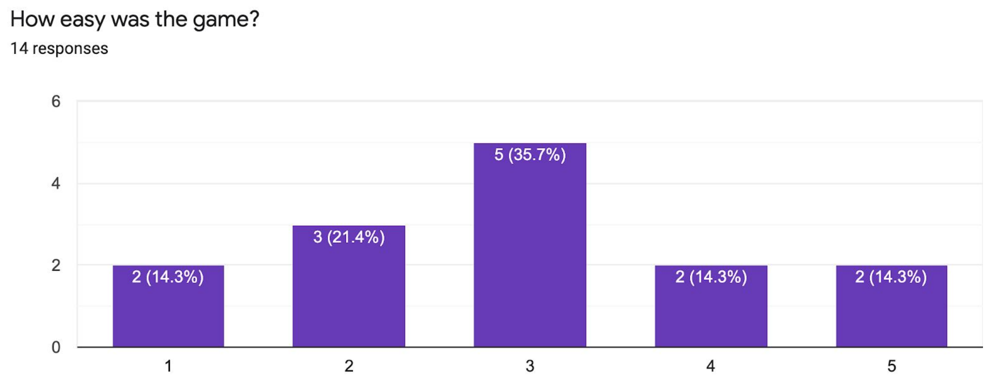
To measure success, we sent out an anonymous feedback form to our COS 426 classmates and some friends. In the form, we asked them to rate the appearance of our game, the enjoyability of our game, the difficulty level, and understandability of the instructions from a 1-5 scale. Below are the results we collected.



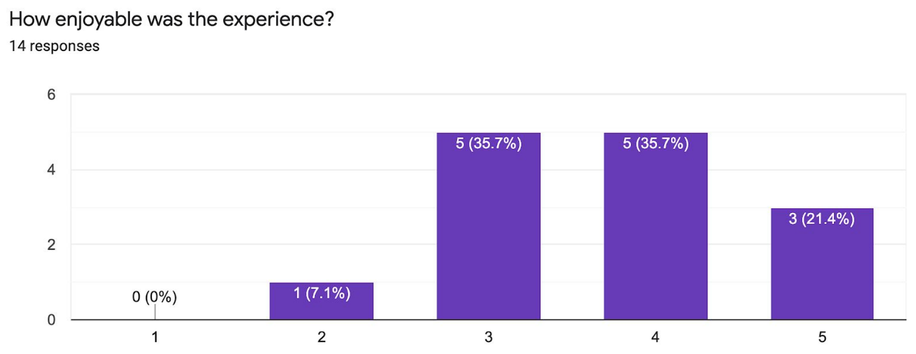
Our game's appearance received good reactions, with most people giving us a 4 out of 5.



Our game's instructions received mixed ratings. It seems that some classmates were more confused about the instructions while others found the instructions easy to understand.



Overall, our game seemed to lean more towards the easier side, though on average people found the difficulty level not too hard or too easy.



Lastly, our game received pretty positive reviews in terms of enjoyability. It seems most people would give the game at least a 3 out of 5.

We received a lot of written feedback as well, and common remarks were about the good character design and the overall theme of the game. It seems that our game is a bit laggy though, especially when multiple obstacles are on the screen and there are also some situations in which the player is unable to attack fast enough to defeat multiple obstacles in a row. Overall, our game has received very positive ratings.

Discussion

In general, we were able to accomplish our basic goals of creating the Avatar character, creating the elements and the obstacles, and making a playable game. We were able to successfully use Blender for character design because there wasn't a previously made Aang mesh, and we also found a good particle engine and other resources for more realistic fire and water objects. Basing the game on a theme was very good for us, because it made it easy to make decisions about the appearance and the pieces of the game. For example, we came up with good ideas for the element-obstacle relationships, and we were able to link each obstacle with the right bending element in the final game. Unfortunately, due to the large number of objects we included in the game and their level of detail, it turned out to be very laggy at some parts. Most of the meshes we included in our game had multiple texture maps applied, and the other objects were imported (mountain, avatar) and were very large. We believe this is a tradeoff for making the game look more realistic rather than cartoonish.
Simpler and smaller meshes would make the game a lot more playable and enjoyable, in addition to other optimization improvements such as removing objects completely from the scene (i.e. with their textures and materials). Although we aimed to have realistic representations of the elements and obstacles, they were still not as good as our initial hopes for the project. We believe the game would look better if we could have more complex representations of fire, water, earth, and air.

Follow-up Work

One of the major pieces that could use more work is the earthbending-gap interaction. As discussed in the Gap obstacle above, we ended up using collision detection to work with the Earthbending object, but this basic implementation does not account for matching the length of the gap with the Earthbending object itself. Another area of work is toggling the difficulty of the game, which we did not really focus on. Currently, you just keep playing by defeating the obstacles, but we could increase the difficulty by increasing the obstacle and runner speed. We also would like to work on randomizing the big rock's position so that it is not always in the center, as well as add small rocks in random positions on the ground for the player to avoid in addition to the obstacles. Some other details we would also add are sound effects for bending elements and collisions, improving the visuals of the all of the objects, and making the collisions more interesting by adding interactions like melting or smoke.

What We Learned

We mainly learned how to use the THREE.js graphics library, especially how to handle parent and child relationships within the modular structure. We also learned how to work with objects and other imports of resources from the Internet. In addition, we learned to use specialized 3D software such as Blender to create 3D meshes. Lastly, we learned about the importance of optimization and saw firsthand how it affected the player experience, something we will remember and work towards in the future.

Conclusion

In conclusion, we successfully met our goal of creating a fun game with interesting graphical elements. Although we are still in the process of improving performance to reduce lag, which would make the experience more enjoyable on older computers with reduced graphics capability, we have created a functioning, aesthetically pleasing game. A few issues were brought to our attention about how sometimes obstacles will appear and then disappear, which may cause the game to end even when there are no visible obstacles; we are currently working on fixing this. Given more time, we would like to add more interesting elements and scenery in line with our inspiration. One of our stretch goals was to add additional characters from the cartoon's storyline as selectable character options for the player. Due to the lack of usable 3D models online for these characters and the time-intensive nature of replicating the look of the base character, adding clothing, rigging, and animating the model, this was not feasible during the project timeframe. However, this is definitely something we would include as a future step. In addition to bug fixes and performance optimization, we would also like to add additional special element-bend powers to extend the parallel to the storyline of the series.

Contributions

We met every few days to talk about our progress on the project and split up tasks for the project. Jingwen took charge of creating and animating the Avatar, creating the airbending and earthbending elements, and creating the rock obstacle. Lily worked on the keyboard controls for moving the Avatar and allowing the Avatar to bend the elements, as well as the moving "infinite" ground, the fire obstacle, and the loading screen and background music. Catherine set up the basic scene elements, added the scenery and background, handled collisions between objects, and worked on the general gameplay logic along with the accompanying HTML/CSS for the landing page and game over page. She also created the firebending and waterbending elements and the ice and gap obstacles. Overall, although we each took control of specific elements, we all contributed to each of the parts of the game and supported each other when we needed help with imports, writing code, debugging and anything else.

Works Cited

Tutorials
https://tympanus.net/codrops/2016/04/26/the-aviator-animating-basic-3d-scene-threejs/
https://gamedevelopment.tutsplus.com/tutorials/creating-a-simple-3d-endless-runner-game-using-three-js--cms-29157
https://tympanus.net/codrops/2019/10/14/how-to-create-an-interactive-3d-character-with-three-js/
https://www.donmccurdy.com/2017/11/06/creating-animated-gltf-characters-with-mixamo-and-blender/

3D Software for Avatar Creation
https://www.adobe.com/products/fuse.html
https://www.mixamo.com/
https://www.blender.org/
https://github.com/the3dadvantage/Modeling-Cloth

Particle Engine
https://github.com/stemkoski/stemkoski.github.com/blob/master/Three.js/Collision-Detection.html
https://github.com/stemkoski/stemkoski.github.com/blob/master/Three.js/Particle-Engine.html
https://danni-three.blogspot.com/2013/09/threejs-particle-engine.html

Objects
https://free3d.com/3d-model/mountain-6839.html
https://github.com/yomotsu/VolumetricFire

https://github.com/mattatz/THREE.Fire

Loading Screen
https://codepen.io/julesforrest/pen/oNvzEgy

Movement
http://learningthreejs.com/blog/2011/08/17/tweenjs-for-smooth-animation/

Textures
https://3dtextures.me/2018/02/27/snow-002/
https://3dtextures.me/2020/01/14/rock-038/
https://3dtextures.me/2018/01/04/ice-001/

Font Files
https://github.com/potyt/fonts/tree/master/macfonts/Avenir%20Next