Froggin

Nicolas Nee, Tyler Benson, Ayo Oguntola, Ryan Hoffman

Abstract:

Froggin originally started as a third person game where you control a frog and have to jump from lily pad to lily pad in order to reach new highscores. As we continued developing, we decided to add a first person mode, where you become one with the frog and can control its tongue in order to capture flies. The goal of the game is simply to reach new highscores. Your score is increased by reaching further lily pads and eating flies!

Introduction:

Goal:
We wanted to create a somewhat difficult platforming game that gave the users an enjoyable playing experience. Froggin is an infinitely generating game in which you travel through various terrains by jumping from lily pad to lily pad in order to reach new highscores. While the game does not get more difficult as you progress, it feels more stressful because of how long it has taken you to get to that point. Our game is one of those frustrating but rewarding games like *Jump King* and *Crossy Road*.

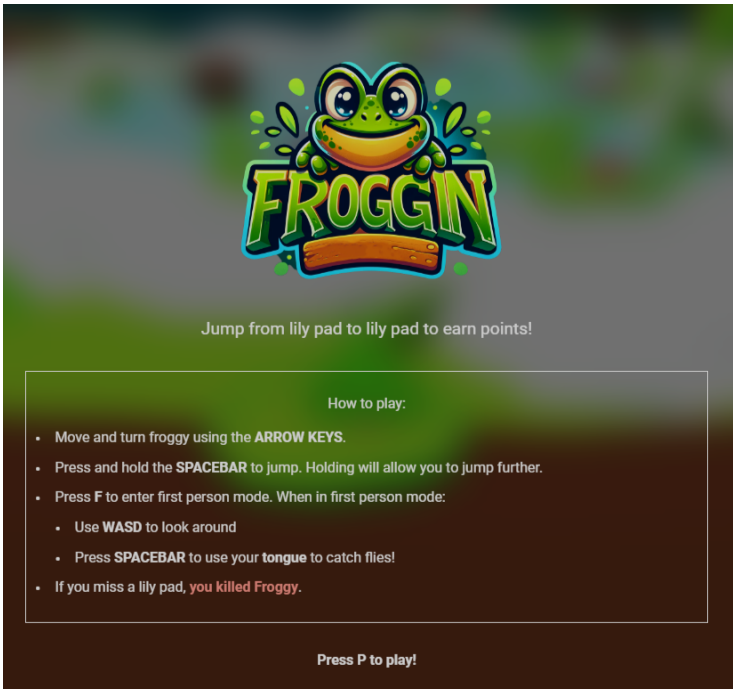


Figure 1: The Froggin Title Screen!

Previous Work:
The majority of our game idea came from *Jump King,* a relentless 2D platformer where you control a king who can only jump from area to area. The game plays vertically, so if you miss a jump, you fall down and lose a ton of progress, sometimes even landing at the beginning of the game. We adapted this game to 3D, where our character was a frog that could jump in any direction to make forward progress. We recognize that our game most likely won't get the same

attraction as *Jump King*, where players fight through the frustration in order to continue in the game, so we added a toggle where the player could see the distance that the frog would be jumping. This makes it much more beginner friendly and does not require hours of playing to learn how to judge the jump distances in order to get far.

We wanted to have a punishable game like *Jump King*, but since we were making the game in 3D, it would've been really hard for the frog to land on a lily pad as it was falling, so we chose to adapt it to be like *Crossy Road* and make progression be more linear. The only purpose in *Crossy Road* was to reach new highscores and compete against your friends, just like our game.

Approach:
We started off by using the template that was provided to us by the staff and built upon it so that we could control the camera manually with the arrow keys instead of it constantly rotating and allowing the mouse to move it. We used different Three.js functions and key event handlers to control the frog's movements and lock the camera onto it. We also had to create a lily pad generator that created new lily pads everytime a new lily pad was reached so that the player had an infinite number of lily pads to jump to.

Our approach to handling physics was a very rudimentary one. Using a full physics engine and external package would introduce a large amount of computation that wouldn't be worth it. The only physics we care about is getting Froggy to feel gravity, collide with lily pads, and fall to its demise.

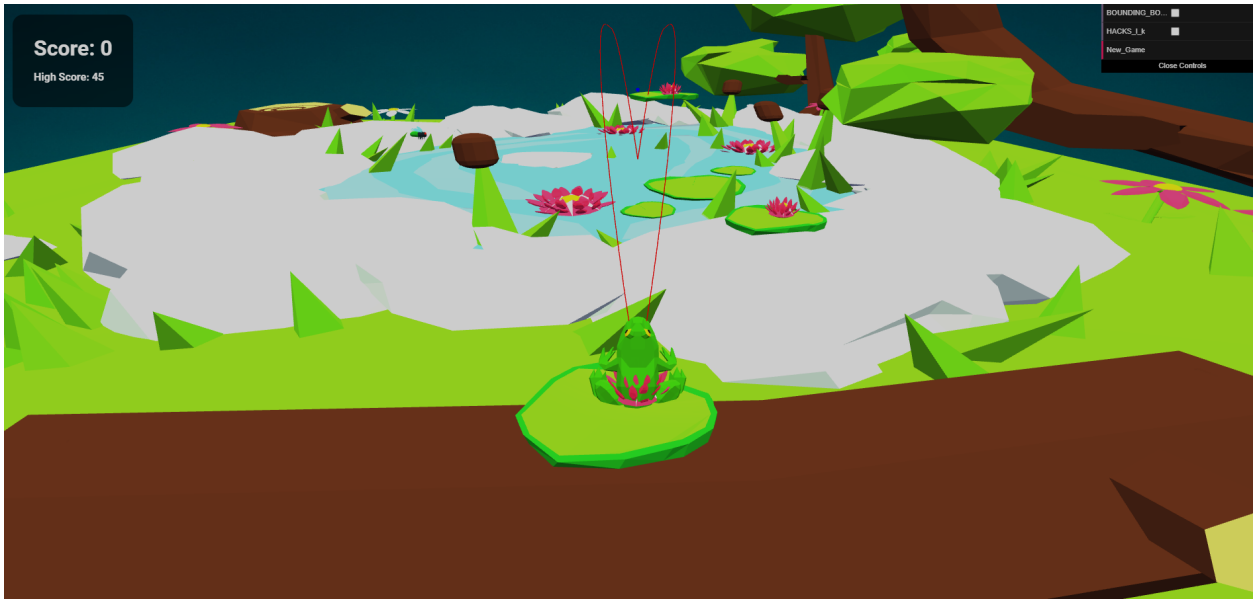


Figure 2: Real Gameplay! In this screenshot, you are playing on the grass pond scene. The red lines indicate how far you are jumping.

Methodology

1. Gravity
We had to implement gravity to be able to correctly allow Froggy to jump. There are multiple possible implementations. We could model Froggy like a particle in a cloth like in A5, such that it experiences a force that is reset and calculated every frame. This was tried, but quickly we realized that in order to jump we would need to simulate a couple frames of force in order to get

enough momentum to actually go anywhere. Also, gravity and jumping are the only forces that act on our Froggy. We decided on a simpler method. Simply give Froggy an upward (and forward) velocity based on how long the SPACEBAR was held down and then, while the Froggy is in the air, apply gravity by subtracting from the velocity in the y-direction every frame. To do this, we include a this.onGround boolean to set when the Froggy is in the air and thus when to apply gravity.

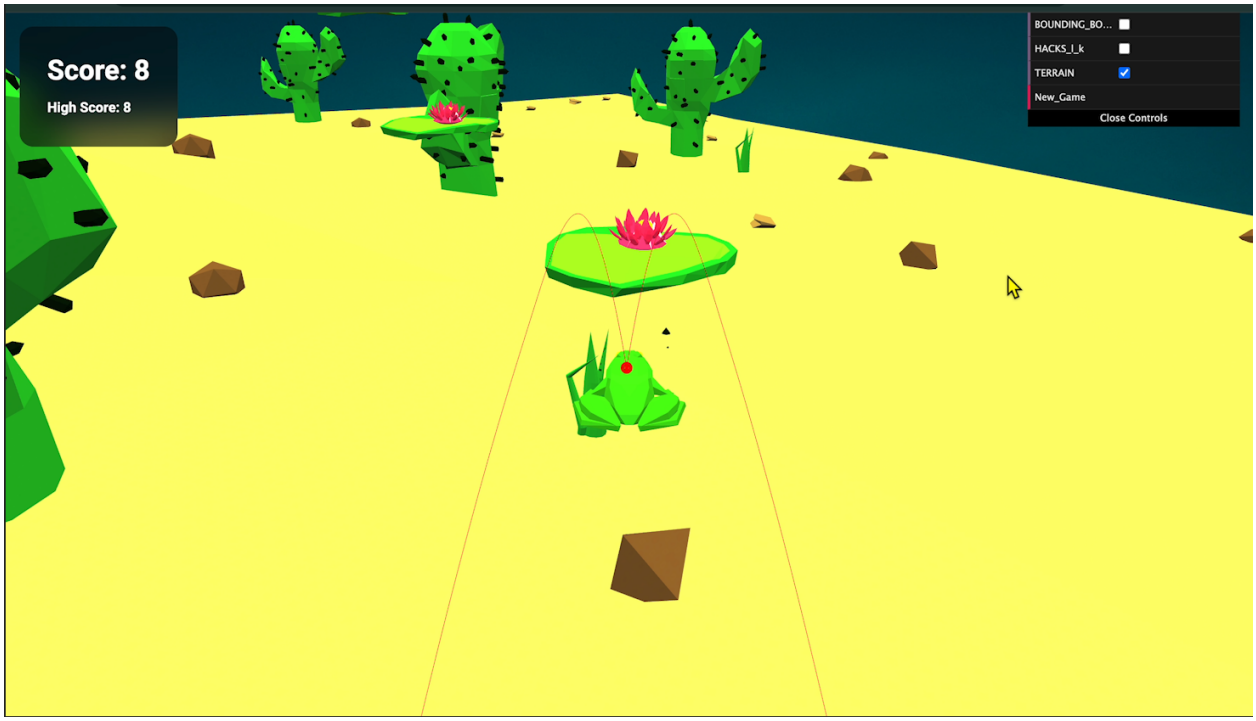


Figure 3: The frog mid-leap on the desert scene. It seems like he is about to die.

2. Collisions

In the same vein, while the Froggy is jumping, we need to check that it collides with a lily pad. Initially, we would get **every** lily pad in our scene and, while the Froggy is in the air, check if the Froggy has collided with any of the lily pads. To check collision, we create bounding spheres in world coordinates and then use the ThreeJS getSphereIntersection function to check for intersections. However, we only want to stop the Froggy when it is *on the surface* of the lily pad (when the Froggy's y-position is less than or equal to the lily pad's) and moving in a negative y direction. When these conditions are met, we set the onGround boolean to true to stop gravity from affecting our Froggy and reset its velocity to 0, essentially colliding with the lily pad. This implementation had some disadvantages. Notably, as the player gets further and further in the game, more and more lily pads were being generated. Checking every single lily pad in the scene for a collision with our frog *every single frame* while the Froggy is in the air resulted in slowing down the frame rate pretty badly. So we optimized our collision detection code to only check **within two lily pads in either direction** (forwards and backwards) of the current lily pad that the Froggy is on.

Also, because our lily pads are generated with a random positive y perturbation this implementation can result in some interesting collision detections where the frog doesn't jump high enough. The Froggy would sometimes collide below the lilypad and be moving in the negative direction – triggering the onGround script such that the frog is reset to the y position of the lily pad and clips upwards. A solution to this would be to make the bounding box of the lily

pad a more accurate object, such as a cylinder of small height. However, this glitch didn't happen too often and made the game a bit more forgiving and so we kept it.

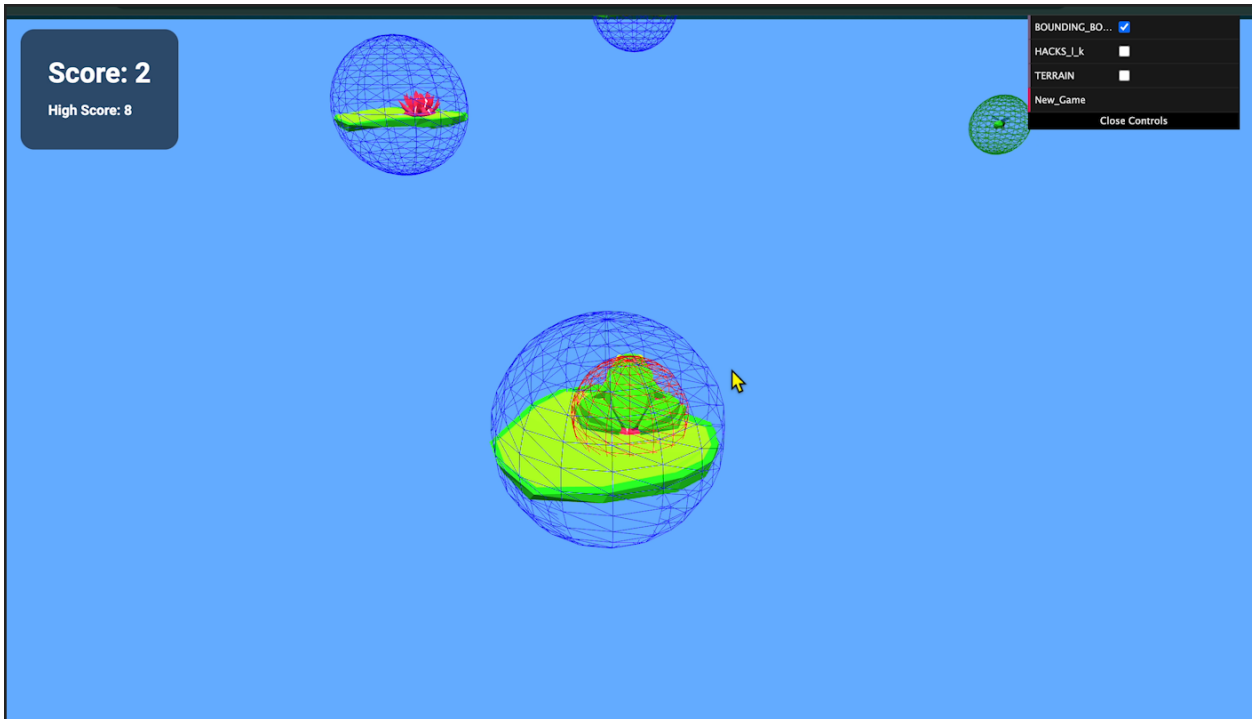


Figure 4: The bounding boxes of a frog landed on a lily pad indicating a successful collision between the two

Figure 5: The frog landed on a lily pad indicating a successful collision between the two in the context of a scene

3. Tongue
The tongue is a separate object, but it is stored as an instance variable of the frog object. It is primarily a pink rectangle, which when the extend or retract functions are called, changes in size based on a preset tween to shoot outwards. The extend function has a directional argument, which specifies in what direction the tongue should shoot (relative to the frog's current position), and it is always called using the frog first person dot position. In each step of the tween, the tongue changes in both its z scale and also in its position (because it scales outwards from the center, but we want the tongue to feel like its coming from the frog and shooting AWAY). On completion, the extend tween called retract, which reverses the motion and then clears the tongue afterwards.

Originally, we implemented the tongue as a line object, which was more convenient because a tween can define the line from points, and so the tween would interpolate the original position of the line with one defined by the endpoint. However, the line was too skinny to look proper, and so we had to scrap this approach in favor of something with some more volume to it.

In order to handle tongue collisions with flies, we create a raycast with origin at the camera position and direction of the dot on-screen. This is better than actually associating the collision with the tongue object itself, because the tongue object moves in space at an angle and has relatively small geometry, so using the dot as a reticule is more accurate. We tried to implement the flies being "sucked" into the frog's mouth by creating a tween that would affect their movement before causing them to disappear, but we didn't end up having time to implement this, so we instead cause the flies to be removed from the scene and the score to increase upon deletion.

Figure 6: The frog's tongue projected projected from the frog's position out towards the aiming reticle

4. Lily pad generation / Moving lily pads
We wanted to have a system that controlled the generation of lily pads as you progressed through the game. We decided to create a generator object that would have all generated lily pads as children. With this infrastructure it would allow us to better track the progress of the frog

throughout the environment, as well as have more granular control over how, where, and when the lily pads would generate.

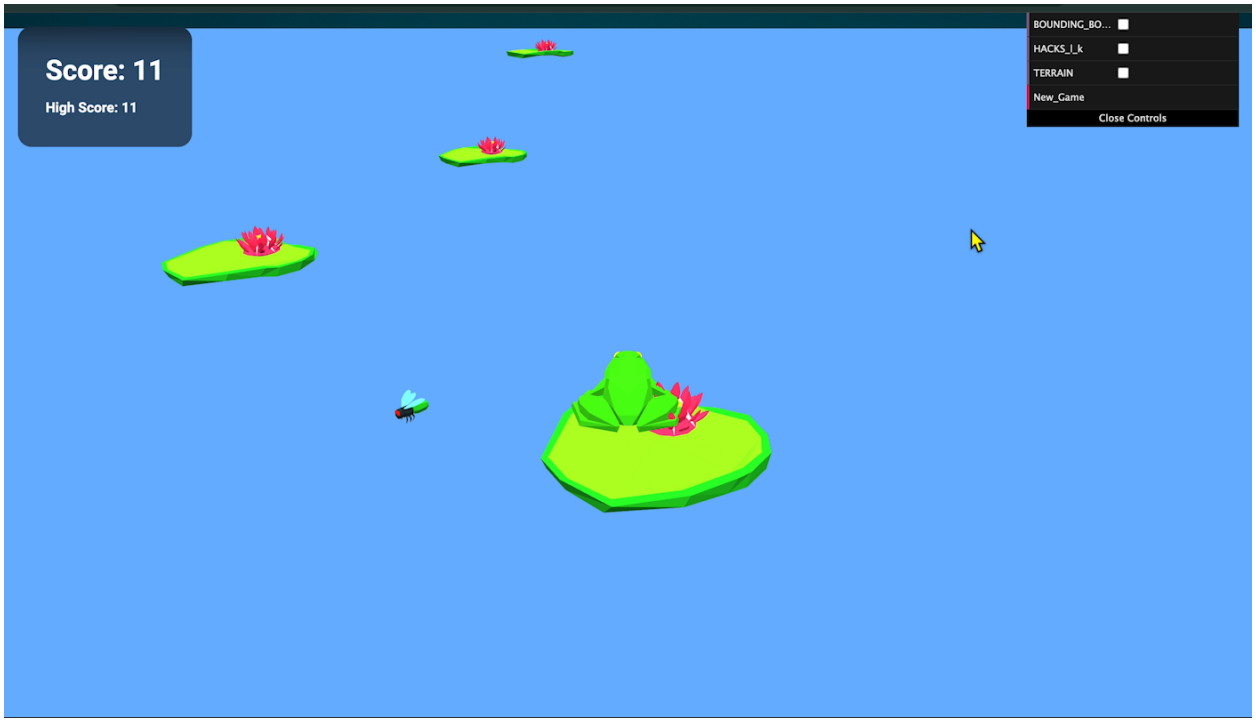
The data structure we landed on was the following: Every generator has a “current”, “previous” and “next” property that store lily pads. The “current” property tracks the lily pad that the frog is currently on and is of type LilyPad, the “next” and “previous” properties are arrays that store all future lily pads currently in the scene and all previous lily pads currently in the scene respectively. This architecture resembles that of a modified linked-list and allows for us to create a sequence of pads within the scene. This allows us to efficiently find nearby lily pads as well as efficiently generate them.

We generate new lily pads by determining a maximum distance it can be from the previous pad with some randomness applied by subtracting by (max_radius / 4) and bounding the angle offset.

The moving lily pads are implemented using tweens that animate the lily pad between an offset of +5 and -5 along the z-axis.

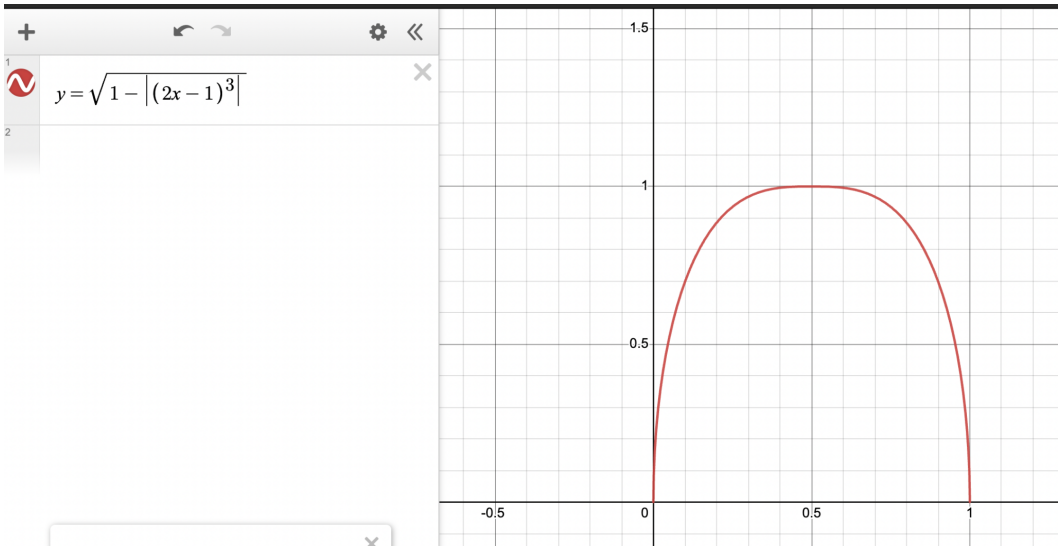
5. Movement, Perspective, and First Person Mode
When I first tried moving the frog, we simply added values to its x and z components, which meant it could only move along the axis. For our game, however, we wanted it to be able to rotate in any direction. I created event handlers for the different arrow keys and set that whenever the left and right arrows were pressed, frog.rotation was changed by the degrees I wanted it to rotate by. For the forward and backwards movement, I had to multiply the distance I wanted the frog to move by the sin or cos of the total rotation of the frog in the scene. I ran into various issues where the frog would move the wrong way, so I had to add different constants and change the math a little bit to achieve the desired result. I think this problem was due to the difference in the world and local positions.

The perspective was a little more difficult than I had expected, mainly because I kept trying to set the camera to the frog's position, without converting the position to world coordinates first. Once I figured this out, the perspective for 3D was quite simple. The camera position was set to the frog's position plus some constant vector that we set for the third person point of view, and the camera was set to look at just in front of the frog. The process for the first person point of view was very similar, where I added a constant first person point of view vector to the frog’s position. For the look at position, however, since we wanted to have the frog eat flies with its tongue, we had to create a sphere when in first person mode that the camera was locked on. If the player switched back to third person, the sphere was removed.



6. Aim Guide
For the aim guide, I created a separate object called aimGuide, which would become a member of the SeedScene scene instead of the frog itself, so that it could more easily be passed world coordinate arguments without having to translate between frog local coordinates.

The aim guide has a couple functions, two obvious ones being clear() and endExtension(), both of which either clear the aim guide from the scene or simply stop its tweening. Then, it has createJumpArcTween and createCollisionArcTween which both help to make the code more readable (although they are both called in startExtension).
Start extension starts by calculating the main parabolic jump curve. It starts by taking the frog position and angle, and creating a set of 500 coordinates which simulate the jump curve. The jump curve itself is modeled after a semicircle equation as follows: jumpHeight * sqrt(1 - Math.abs(2(x-1))^3)). Modeling this equation in desmos looks like:

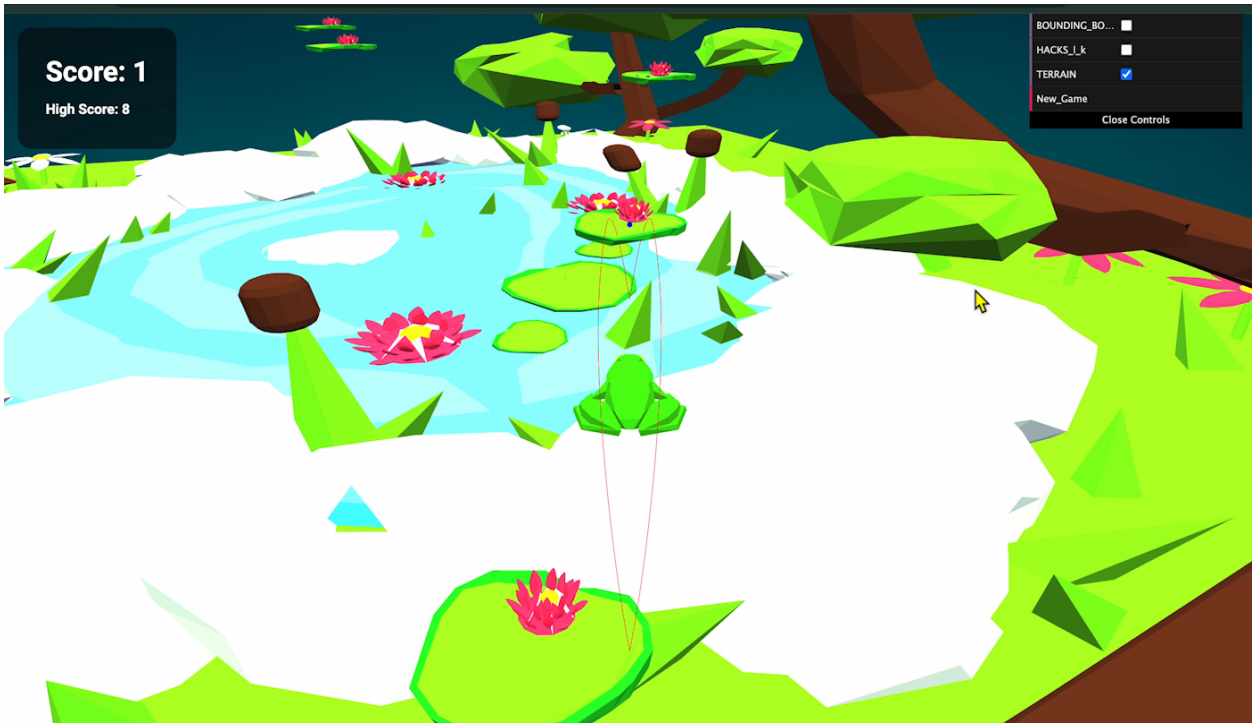


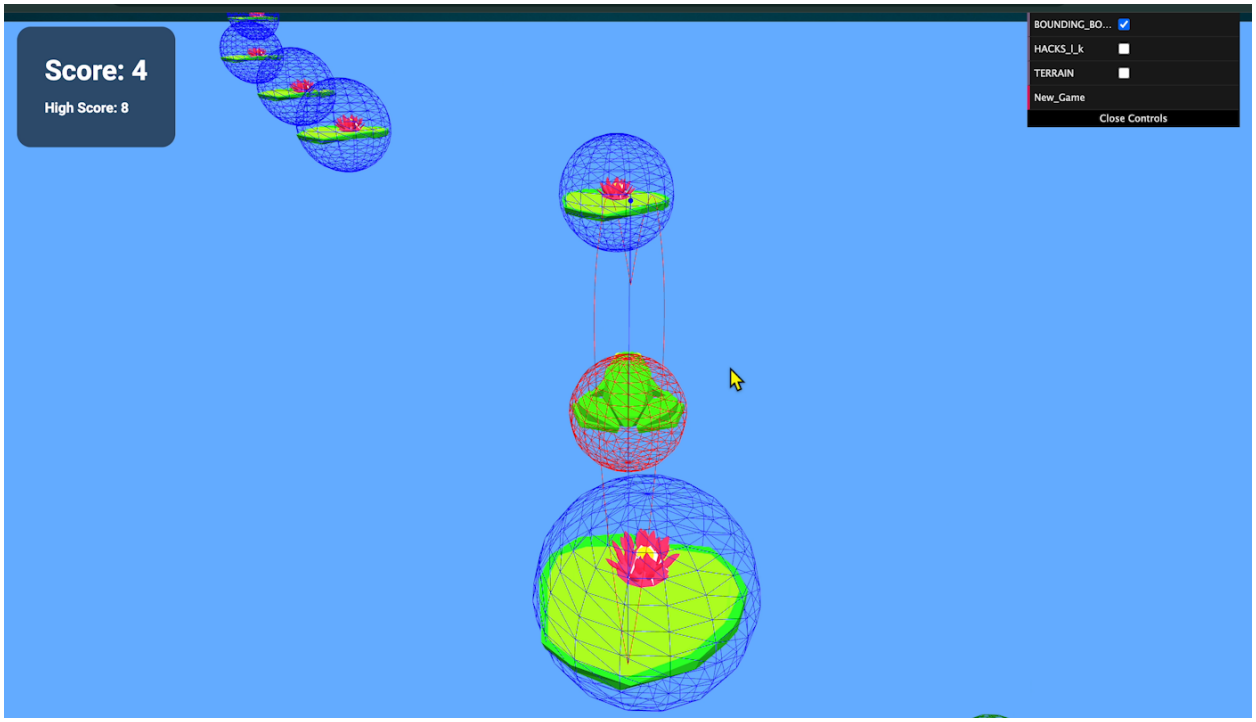
Because of the nature of our jump/gravity mechanics, we don't have access to a clear motion equation which can perfectly predict our jumps. Therefore, this function was derived using a lot of trial and error and visual inspection. If given more time, this would definitely be a place for improvement, as perfectly simulating the jump would be just all that much more accurate. Regardless, this function works quite well.

I create three of these lines, and use a quaternion to rotate two of them slightly to the sides. This is because in our perspective where the frog always points in the direction of the camera, these parabolic lines look like straight lines when head-on, and therefore need to be tilted in order to be fully visible.

There is one hidden line that is not tilted though, and this line is used to detect collision with lily pads. In each step of the tween's growth, the middle parabola calls a check collision function, which uses the lily pads argument to inspect all of the lily pads and decide if the point is inside them and at a similar y value to the pad's surface. Importantly, the function only checks points which are in the latter ⅔ of the arc, to avoid finding collisions with the current lilypad (which can happen when some of the geometries aren't as perfectly aligned as one might hope). If the collision is found, the usual circular target disappears and a temporary target appears on the pad's surface where the jump will hit the pad. This helps indicate to the player when their jump will work, and helps give them a sense of the power behind their holding of the spacebar.


Score: 7
High Score: 8
BOUNDING_BO...
HACKS_I_k
TERRAIN
New_Game
Close Controls


Score: 1
High Score: 8
BOUNDING_BO...
HACKS_I_k
TERRAIN
New_Game
Close Controls

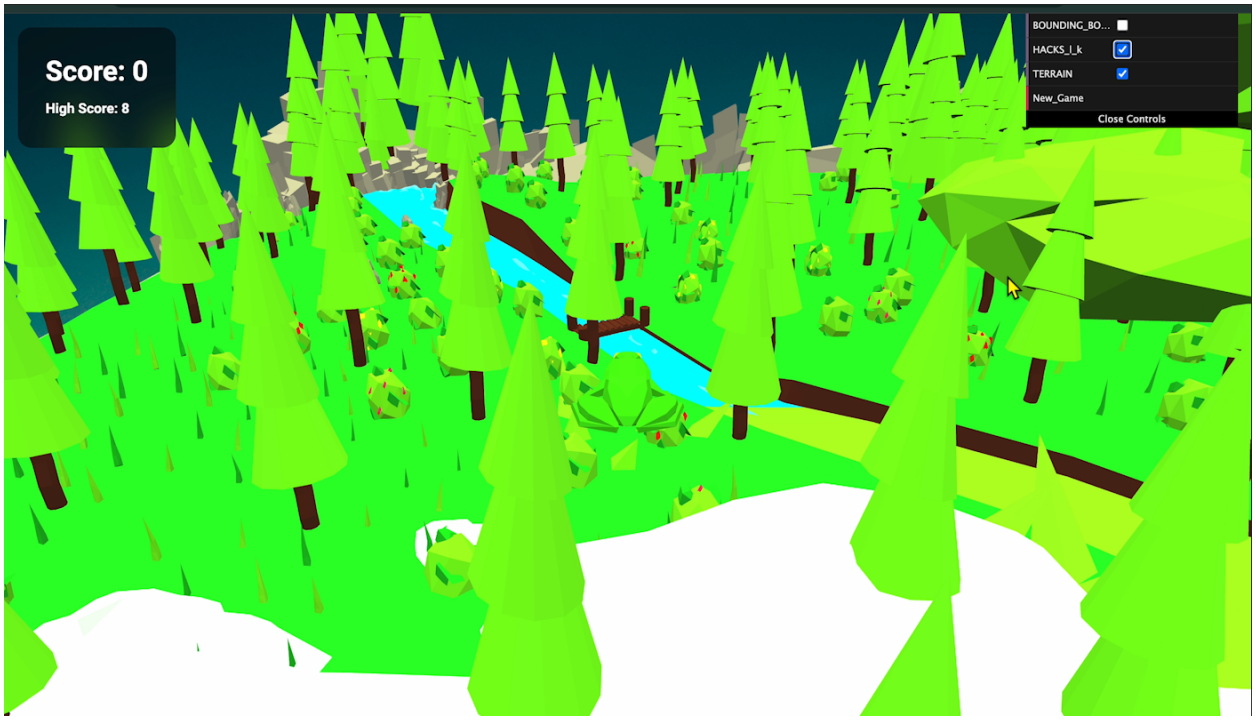


7. Audio

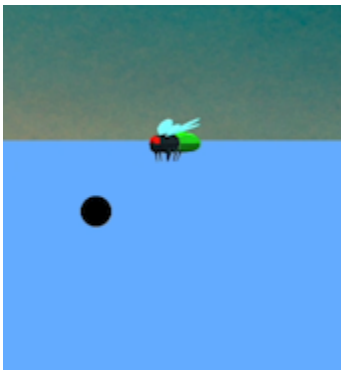
We implemented the audio using the different audio related packages from Three.js, such as AudioLoader, Audio, and AudioListener. For each sound we added a listener, a sound, and an audioloader. We loaded in the audio from our raw.githubusercontent url and set the sounds parameters, such as the volume or whether it looped once it was finished. For the jump, land, and death sound, we did not want them to loop, but we did for the game music. The music would start when the user pressed p, but if the user pressed p multiple times, the music would stack on top of itself and sound horrible. To fix this, we added a boolean variable to check whether the game had been started. If it had not been and p was pressed, we played the audio and then set the boolean to true so that the sound would never play again.

8. Infinite Scene Generation

The actual code for the scene generation was pretty easy, the only annoying part was making sure that the scenes all spawned at the correct spots. To check whether a new scene needed to be added, I simply checked whether the frog's euclidean distance from the scene's spawn position was greater than some value. If it was, then I added a new scene. I saved each generated scene in an array, so that I could check the frog's distance from each new scene, so that scenes would not be generated on top of already existing scenes. Because of the way that this code worked, however, I needed to make sure that each scene's length was the same as the threshold value for generating a new scene, so they wouldn't overlap. The different scenes that we could use were quite limited because we did not want our frog clipping through any trees or rocks and running the player's immersion. The gui toggle "HACKS_L_K" is meant for the grader to be able to explore this feature. Holding 'L' will move you in the x direction and slightly up, and 'K' will move you straight up.



9. Flies
Implementing Flies turned out to be much more difficult than I thought it would be. I initially added each Fly (generated with 10% chance) to be a child of each LilyPad, but that resulted in whenever a LilyPad moved, the Fly moved with it. So I instead added the Fly to the scene itself, and computed the position as world coordinates. However, I wanted the Flies to rotate at random rotations around its own axis, which proved difficult as their rotations were being computed relative to world coordinates and they were being thrown around the entire scene – eventually crashing my browser. I opted to let the flies simply… not rotate. I'm certain there is a way for me to fix this issue and this is an area for next steps.



Results

Initially, when we had very little progress, we quickly got in the habit of being happy with the little improvements we made. Our success was determined by whether or not we were able to actually implement an idea, especially when we wanted to do so many things. Since our game

is very interactive, everytime we would change or add a feature, we would have to experiment with our game. Test playing allowed us to tell and help debug problems that would arise as we coded. We could collect data from each run to either examine what was breaking in the game or to print out arrays and positions to understand the mathematical errors.

We measured success based off of two primary metrics: reliability and playability. This meant consistency of jump length, lily pad timing, and aim arc among many others. We wanted to create playable game mechanics that a player could reasonably improve with time and practice. Our own high scores within the team were clear indicators that as we played the reliability and playability allowed us to improve, however we got qualitative evidence for this as well by gathering feedback from friends and classmates. These two forms of feedback lead us to conclude that we were successful in our goal and that we created enjoyable game mechanics.

Discussion

Ultimately, we need to rethink our design of our Scene Graph. It would be extremely helpful if each object had a relevant *parent* as well as had easy access to the properties of the scene and other objects in it. For example, when trying to implement positional audio, allowing the Froggy to access properties of the Flies (in order to get the listener object to listen to the sound that the Flies make) proved extremely difficult. There were some cases of logical relationships between objects, but I'm not sure this was always effectively managed. Another consideration was the extent to which we wanted to specifically make the physics of the game accurate. We could have taken inspiration from the Cloth assignment from the class, but instead decided to simplify. This potentially limited the extent of future features, but we believe helped in creating short term reliability.

Conclusion

We think overall we achieved our goals very well. We have found our game to be very fun to play and highly addictive, which is a major testament to our concept. I think we learned a lot about the THREE.js workflow during this project, and ultimately developed our coding skills in a team collaboration setting very well. Some of us became much more comfortable with the git workflow during this process too. We have found that through player testing most people really enjoy the game and it feels incredibly well featured and vibrant, and has reliable mechanics.

I had two really big stretch goals for this project. First, I wanted to make a Stripe Express Checkout Element that would allow a user to pay $1 whenever they died after achieving a high score to return to the lily pad they were previously at before dying. Obviously, this would require a backend that I had setup as a Firebase serverless function that allows communication with Stripe API to get the necessary secrets. Additionally, I had an unreasonable amount of trouble simply trying to get a static file accessible via a url on our site. That is, I needed https://msnxus.github.io/Froggin/.well-known/apple-developer-merchantid-domain-association to respond with a file that tells Apple that we are a known and reliable site such that it allows Apple Pay. Because we already had html content being served when you died, it wouldn't be too difficult to follow the Stripe docs and integrate the Express Checkout Element; however, the simple issue of static content discouraged me enough to give up on it.

Second, I wanted to set up a positional audio element such that whenever the Froggy passed near a Fly (or rotated itself in a circle with a Fly nearby) the audio would play a buzzing sound

positionally relative to that Fly. A super cool idea and something I saw in the ThreeJS examples. However, as described above, I had difficulty sharing properties between these two objects as well as in loading the audio *as static content*, loading it into a DOM element and then accessing it via the Fly. I got both of these issues to work; however, there must have been some issue with world vs local coordinate systems and thus the audio would only play positionally from (0, 0, 0). Unfortunate.

Contributions

Tyler: jumping, collisions, menu/death html pages, flies, score/high score
Nico: movement, camera work, first person mode, audio, infinite scene generation
Ayo: lily pad generation, lily pad variations, html infrastructure, audio
Ryan: first person aiming reticule, tongue, aim guide (jump visualizer lines), water

Works Cited

- *Jump King*: https://en.wikipedia.org/wiki/Jump_King
- Crossy Road: https://en.wikipedia.org/wiki/Crossy_Road
- Game Music from https://www.zapsplat.com
- Sound effects from https://freesound.org/
- CAVE -> Cave Scene | Draft 1 by Danni Bittman [CC-BY] (https://creativecommons.org/licenses/by/3.0/) via Poly Pizza (https://poly.pizza/m/2VmWkpO6OvK)
- ROCKPOND -> Pond by Poly by Google [CC-BY] (https://creativecommons.org/licenses/by/3.0/) via Poly Pizza (https://poly.pizza/m/5rf3YuZfJAW)
- GRASSPOND -> Pond by Jarlan Perez [CC-BY] (https://creativecommons.org/licenses/by/3.0/) via Poly Pizza (https://poly.pizza/m/1XazZeNBWG_)
- WOODS -> Nature by 3Donimus [CC-BY] (https://creativecommons.org/licenses/by/3.0/) via Poly Pizza (https://poly.pizza/m/0nsE2b8uXZy)
- DESERT -> Desert by Poly by Google [CC-BY] (https://creativecommons.org/licenses/by/3.0/) via Poly Pizza (https://poly.pizza/m/a1HnTCHfE34)
- Frog -> Frog by Poly by Google [CC-BY] (https://creativecommons.org/licenses/by/3.0/) via Poly Pizza (https://poly.pizza/m/0QZHgL_V-Pj)
- Lily Pad -> Lily Pad by Jarlan Perez [CC-BY] (https://creativecommons.org/licenses/by/3.0/) via Poly Pizza (https://poly.pizza/m/2SPd7jqCmPF)
- Fly -> Fly by jeremy [CC-BY] (https://creativecommons.org/licenses/by/3.0/) via Poly Pizza (https://poly.pizza/m/f8kM9xA_5sV)