Pool Whirl

Introduction

Our group created an infinite single-player game called PoolWhirl. The player's objective is to navigate a teddy bear on a tube within a whirlpool. The teddy bear must avoid obstacles floating on the water and collect as many points as possible; its motion can be controlled using the left and right arrow keys. Colliding with rubber ducks causes the bear to be knocked toward the center of the whirlpool and the player to lose points; ducks also become haloed and super-charged (more damaging) after lightning strikes. The player gains points by consuming fish. The game ends when the teddy bear is sucked into the center of the whirlpool.

Previous Work

Pool Whirl was primarily inspired by car racing games wherein a car races around a racetrack; in like manner, the teddy bear rotates around the whirlpool. Our point and obstacle system was also influenced by *The Aviator*.

Approach

Our major goals were twofold: 1) to simulate the whirlpool's pulling force to and a buoyancy effect and 2) to handle collisions/interactions between all objects and fine-tune the individual positioning and movement of the teddy bear, ducks and fish within the whirlpool. We aimed for a minimalistic, cartoon aesthetic; this approach guided many of our implementation decisions for the whirlpool and objects, discussed below.

Methodology

Whirlpool:

The whirlpool was initially implemented with the Cloth assignment. Particle.js was used to make random impulses on the surface of the cloth. The water force applied a sinusoidal force to random particles. Centripetal force and gravity were applied to the whirlpool, but gravity made it difficult to keep the cloth in place. A velocity vector was added, but this led to a strange tearing of the cloth and its disappearance. A fake centripetal force was applied to the particles by using sine oscillation inversely proportional to their distance from the origin. The program crashed because the particles rotated at different speeds. Furthermore, the plane structure didn't achieve the goal of depth.

Instead of the cloth simulation, a geometric cone shape became the basis of the whirlpool, where it was rotated along its y axis upside down. A method was created that randomly selects a vertex index in the cone mesh, as well as a random scale (how far up and down the pixel oscillates),

stores the original y position of the random vertex (so it oscillates around it's original y position), and a random timescale (so that each vertex oscillates at a different time). With all of this combined with rotation, it made a convincing whirlpool. Additionally, the cone's rotation is increased with time. Finally, the amount of radial segments in the mesh was reduced to make it look more low-poly.

Although the water was made to look like it was being sucked into the center of the whirlpool, the vertices of the cone actually do not move outside of up and down. In order to simulate the pulling force of a whirlpool, we could not rely on physics from the whirlpool vertices like we were able to do in the cloth assignment. Instead, a small translation was made on the teddy bear that was called with each render. The translation must be less than the translation imposed on the bear when the player moves the bear in the opposite direction. Additionally, whirlpool colors were generated by saving colors into an array, and assigning each respective element to each vertex. Every third element was made darker, with some light dashes added into the whirlpool as well. Reflection and refraction was attempted, but an attempt to implement Fresnel shading did not work on the cone, and additionally, it was difficult to map the environment onto the cone when the environment was dynamic.

Scene:

The clouds were made using individual sphere geometry objects, and they become darker during the duration of each lightning flash, only to start over white again. They start flaring blue with the implementation of a flashing point light when the lightning is about to strike. Additionally, two directional lights are used to color the scene, as well as an ambient light. The color of the sky changes between lightning flashes as well.

Lightning:

To generate lightning, we use an L-system as specified in this tutorial. We initialize a list called segmentList that will eventually contain all our lightning segments, and begin with a line segment defined by startPoint and endPoint. On every generation, we iterate through each of the current line segments and 1) discard the existing line segment (after saving its start and endpoints) 2) subdivide it, offsetting its midpoint by a small random offset and 3) adding these two new segments to segmentList. At the end of each generation, the maximum allowable offset maxOffset is halved. We also create lightning branches during every other iteration by adding three segments to segmentList instead of the usual two; the third segment continues roughly in the direction of the first, with some randomization thrown in. Initially, we tried to make the lightning target an obstacle to avoid. However, the frame rate slows down too much when lightning is generated, making that infeasible; instead, we paused scene rendering—so that the entire scene comes to a standstill—and also made the duck haloes (see **Additional Notes**) visible upon each lightning strike.

In order to make our lightning glow and illuminate the objects in our scene during the storm, we use an effect composer for postprocessing. We first add a render pass that will render the scene with the camera into the first render target. Next, we add a bloom pass, which applies the glow.

Lightning Functions

We add all our storm objects directly to the main scene and set their .visible property to false in order to hide them. To begin, lightning_init(startPoint, endPoint) is called to create the lightning given a set of start and end points. activate_lightning(), which activates the lightning animation, is called within our render() loop. To control the timing of our lighting animation, we have two global variables, lightningTimer and lightningInProgress. If lightningInProgress is true, we immediately return from activate_lightning(). Otherwise, the function sets lightningInProgress and the visible property of all storm objects to true, the scene background to black, and the effect composer's renderToScreen property to true. Then, it initializes lightningTimer to 5. Lastly, it calls requestAnimationFrame(animate_lightning). animate_lightning decrements lightningTimer, makes the lightning visible every other frame to create a flickering effect, and calls composer.render to display the glow. Once lightningTimer reaches 0, lightningInProgress is set to false and deactivate_lightning() is called; otherwise, the animation continues with requestAnimationFrame(animate_lightning). deactivate_lightning()simply sets the visible property of all storm objects to false, the scene background back to blue, and the effect composer's renderToScreen property to false, then calls composer.render() again to deactivate the glow.

Animals:

The teddy mesh consists of many boxes, and two functions oscillate the ears and head respectively. The function TeddyMunch() rotates the teddy's arm up and down by a certain amount a certain amount of times in the render() function as long as it's "teddyTime", meaning that the animation hasn't finished oscillating the teddy's arm fully back and forth.

We created meshes for our fish and ducks using different shape geometry classes from three.js. Using a triple for loop, we iterated through an interval of points that were contained in the smallest box that contained the whirlpool cone. If the point is inside the cone, a duck would be created at that spot, pushed to an array containing all the ducks, and be made invisible for later.

However, the above approach was computationally taxing since it included checking points that were not on the whirlpool and computing to see if each point was on the cone or not. We suspected that the number of invisible ducks was making our game lag, so we hard coded in 5 ducks with set positions. We wanted more ducks, so we found a more efficient way to place ducks. Next, we iterated through all the vertices on the cone, and created a duck at the position of every 8th vertex. This means that we are only checking positions on the cone, and we take out computing to see if a point is on the cone.

The method used to create obstacles and point objects involve instantiating all ducks and fish at the beginning and setting all of them to invisible. After a predetermined amount of time, a random duck or fish from the existing arrays is made visible. The ducks stay visible so that

eventually, all ducks will be seen and the player will have to attempt to evade all of them. This is meant to increase the difficulty of the game as the player survives longer. The haloes on the ducks are initialized within ducksInit(), set to visible within the activate_lightning() function, and deactivated 20 seconds after the lightning is. While the haloes are visible, the ducks are "charged," or more potent; colliding with haloed ducks costs the player more points. The fish are made invisible once the bear eats it; however it is still in the fish array that is being accessed to make the fish visible, so it will eventually be randomly picked to be visible again to be eaten.

Our best version of buoyancy is what you see in the program. The big problem with buoyancy is the hassle of finding the closest vertex. In order for the mesh to be above the water, we thought it would be a good idea to find the closest vertex in the mesh, but you would have to go through all the vertices in the mesh to find the closest vertex. You only have to do that once for the fish and ducks, but with every frame for the teddy. This is because the cone is rotating under the teddy, so the vertex positions relative to the teddy are constantly changing. For the duck and fish, even using a closest vertex produced unpleasant results because the vertices near the duck and fish could still change, and positioning the ducks and fish along the cone was already imperfect, so with the oscillations some of the ducks and fish would be too high or too low consistently. We tried to implement gravity with a force of buoyancy upwards using Verlet integration and the formula for buoyancy, however we found that we would get huge values for y, and we were never really able to solve it. Finally, we settled on using a sinusoidal oscillation to find the amount by which we would translate the teddy/duck/fish up and down, at a scale and a timescale that was about half of the maximum scale and timescale of the random vertices (oscillating due to the water noise). It's imperfect, but the ducks and the fish are still collidable. The fish had difficulty with buoyancy although it was implemented in the same way as the ducks, so the fish aren't buoyant.

Gameplay View and Movement:

Positioning the camera properly took some time. At first we tried to get it to focus on the teddy's face as the teddy rotated with the whirlpool, but it's lookAt vector would only look at the side of the teddy, and never moved around with the cone properly. Eventually we decided that it would be easier and more efficient to keep the camera stationary and the teddy not rotating, instead rotating the cone under the teddy.

To position everything, we made the function moveObject, which has event listeners for different keys. We printed the position each time we moved something, and when we settled on a position, we would read the final position and use it as the initialization position for the object. Before we had tried moving everything with individual translate statements, but soon that got annoying and messy. moveObject is now the way that the teddy is moved. We later changed the moveObject function to avoid delay from holding down keys by continuously moving the teddy in a direction while there was a keydown event, and to stop when there was a key up event. This worked quite nicely (credit: Reilly Bova).

Gameplay Collisions:

There were two sets of collisions to handle: duck collisions and fish collisions. Both are implemented the same way, with slight variations on what happens when the collision occurs. This three.js collision code was used in order to handle collisions: https://github.com/stemkoski/stemkoski.github.com/blob/master/Three.js/Collision-Detection.html.

This is a raycaster that casts a ray from every vertex of the tube from the origin. If an object intersects at a distance closer than the distance between the vertex on the torus and the torus origin, it registers as a collision. Using this raycaster was enormously helpful, but since the duck buoyancy is a bit off, the teddy could come pretty close to a duck and still not get a collision. Therefore, we added an EPS that would allow for collisions to be detected more easily.

The fish collisions are also implemented with a raycaster, this time checking for intersection of fish objects rather than duck objects.

When a collision is detected, the loop through the tube vertices is broken, and the sound effect, as well as the visibility of the fish, changes. Further, the points are either brought up or down depending on whether it was a regular duck, a haloed duck, or a fish.

Making the fish invisible was harder because it was a Group. We originally tried to do so by going through all the fish so we could specify a fish to make invisible, but that was inefficient, so we instead tried to get the object of the collision and make that invisible, which only made part of the fish invisible because it was a group of parts, so then I had to trace the parent and make each child invisible.

In addition, we handled sounds for thunder and background music. Background music was difficult at first because the audio context is automatically suspended in Chrome, but we made a function to turn the music on and off with a button, which fixed the problem.

- http://soundbible.com/1378-Quack-Quack.html (duck sound)
- http://soundbible.com/1968-Apple-Bite.html (fish sound)
- http://soundbible.com/2053-Thunder-Sound-FX.html (thunder sound)
- https://www.fesliyanstudios.com/royalty-free-music/downloads-c/happy-music/2 (Pirate Dance, Background Music)

HTML Elements:

We first created a lostGame function that detects if the teddy position is within a certain radius of the center of the whirlpool, and if so, shows a modal on the screen saying the player lost. If the player loses, they can click on the close button or off the modal to play again, which refreshes the window. Then, we created a point system, which put a box in the corner and updated the innerHTML string whenever a point was gained (in a fish collision) or lost (in a duck collision). Finally, I created an intro, which describes how the game is played. We added pictures for the teddy, duck, fish, and haloed duck with a little description of what each does. I borrowed the modal code

(which was found on ww3schools.com) and added a background image. Later, we decided to improve the look by generating separate HTML pages for intro, initial screen, and gameplay, and using the meshes rather than pictures to explain gameplay.

Discussion:

Setting out to create a whirlpool was an ambitious goal: physical properties of whirlpools that seem simple enough don't necessarily directly translate to ease to simulate, as seen in directly trying to use the forces associated with physical whirlpools as a force in cloth, just like gravity. The difficulty of recreating physical forces in a computerized scene is further exemplified in our difficulty with buoyancy, water noise, and collisions. Even without this, accounting for efficiency in creating obstacles, the creation of complex obstacles such as lightning, and the texturization of water, as explored in the methodology section above, can be very challenging. Through the creation of this project, we learned what things could be done and what things must be sacrificed to create a game: a game with movement and objects not necessarily based on physical reality, but rather focused on creating the appearance of a game based on said physical properties. Once we moved away from the mindset of a strictly physically accurate implementation, our game became much faster, with much more pleasant graphics, including taking into account user feedback from classmates and friends.

With all of that said, there are some things that could be improved for the future of our game. The game is rather simple in its current form, some additional features common in other games could most certainly be applied to this one with a little extra time. One example is a multiplayer feature, where there are two bears competing to get limited fish and knock each other to the center of the whirlpool. This would require 2D instead of 1D player control, and a different coloured mesh for the other player to avoid confusion, but otherwise, it wouldn't be too difficult to implement. Another possibility is powerups: the bear might have specialized fish that can make it faster, immune to being sucked in the whirlpool, able to knock away ducks or use ducks as additional points (when winter is coming, a bear will do what a bear's gotta do). Additionally, certain effects could be added to the game to give it a more impressive look. If it could be done efficiently, water foam could be added under the teddy, so it appears like it is skidding through the water. Rain might be added at certain points, or a switch from night to day so that survival points could also be measured in how many days you survived. There are many possibilities as to where this game could go.

Conclusion:

We were able to attain a majority of our goals. We created a working and convincing whirlpool scene and added objects that had different interacting properties. The scenery is playful, reminiscent of a child's play pool and we were able to add audio to further give it a child-like feel. The ducks, lightning, and whirlpool act as obstacles to make the game challenging while the fish gives the player incentives to play (increasing their high score). While we were able to produce a

successful end product, some things did not go as planned. We originally wanted lightning to act as a direct obstacle to the player, but we compromised by having the ducks be more dangerous when lightning strikes. In addition, we were not able to fully model buoyancy, and settled for using a uniform sine wave instead. In the future, we can revisit these issues to model a more convincing model of buoyancy that reflects water movement and have the lightning interact directly with the teddy bear. Our program provides a nice scenery with compelling gameplay through interactable objects.

Contributions:

Emily

- Storm elements:
    - lightning generation, animation, activation, deactivation
    - storm lighting
- Postprocessing/Bloom effect to make the scene objects glow (during lightning)
- Title font and animation using CSS keyframes
- Fish mesh
- Haloes on ducks (mesh design, activation, deactivation)

Nsomma

- Teddy mesh and associated functions
- Final wave/particle oscillations, attempt at using Particle.js for centripetal force
- Collision detection
- Intro layout (Design of the different HTML pages)
- Sound effects

Rachel

- Duck mesh
- Duck and fish randomized generation
- Buoyancy of ducks and fish
- Pull force of the whirlpool on the teddy bear

Sean-Wyn

- Geometric cone whirlpool, initial wave/particle oscillations on cone, camera positioning
- Fish mesh, Tube mesh
- Whirlpool design, reflection/refraction attempts
- Cloud movement/design, background scenery
- Storm elements, point lightning generation