Priscilla Bushko (pbushko)
Peter Chen (pcchen)
COS 426 Final Project

Introduction

For our project, we were inspired to make a game with custom animation and have interactive elements. Our goal was to make a multiplayer game that is a human playing fetch with a dog. This game is intended to be wholesome, fun for all ages, and easy and intuitive to play.

To approach this project, we used a two-player system where one player has the control of the dog and one player has the control of the human. The object of the game is to throw the ball and have the dog return the ball to the thrower. We used a mini-map to keep track of the players in case the dog or ball is out of view. To make the game more immersive we created animations for the dog and added textures to the scene. The game is using a first-person view for the human player, and the dog is using a third person view.

Implementation

Three.js

We used Three.JS, an open-source, lightweight, library for WebGL rendering. This library was used previously in assignments, and this familiarity influenced our choice for rendering [1].

First-Person Camera

For the Camera we used a PointLockCamera system to track the mouse position on the screen and rotate the camera view. The camera movement is translated by using the keyboard keys (WASD) to translate the position of the human player. The view of the camera is rotated using the PointerLock, which is activated when the player clicks the screen and deactivated by using the ESC key [2].

In order to make the camera follow the terrain of the map, we used raycaster to gather intersections from the player to the ground. The height camera is a raycaster that emits a ray from the player's head to the ground, and if an intersection is detected then the player's height moves with the distance to the ground. In addition, we also implemented a height raycaster for the dog as well. This makes sure both the player and the dog do not run through the hills on the terrain.

Further, we implemented used an auxiliary ball to show if the player has the ball or if the dog has the ball. This ball is implemented as a sphere that floats in front of the player's camera, when they have the ball and in the dog's mouth when the dog has the ball. The camera ball's position is locked to the player and the dog respectively by adding it as a child object of the player or dog object.

Mini-map

For the mini-map we created an auxiliary camera system that is positioned high in the sky and pointed vertically down at the game map [3]. The camera is used to set another viewport in the upper right of the game viewport. This mini-map shows the position of the players as well as their direction and the position of the ball. To track the position of the players, we created auxiliary pyramid objects that are rotated to be orthogonal to the mini-map camera. The pyramids are placed above the skybox and cannot be seen by the player camera, but can be seen by the mini-map camera, and are then rendered on the mini-map viewport. The tracking objects for the dog and the ball are simply implemented with the position and rotation of their respective objects. However, for the camera, the position can be set accordingly, but the rotation must be set with the camera's rotation. There is bug with the camera rotation implementation and the mini-map only tracks the rotation if the angles are between 0 and 180 degrees.

Trees

For the trees, we implemented them using the OBJLoader. The loaded OBJs have the material of the model as well as the color and texture. We tried using more complex OBJs, but only the branches of the tree would render and without any texture. In addition, we tried using Blender models of the trees, but this also did not work because we could only import the leaves instead, which were also without texture. We were able to work around these shortcomings by finding a simple and suitable OBJ [4].

Lighting and Skybox

For the lighting, it was difficult to make the lighting in the scene because we have many objects, and if all the objects have shadows then the rendering is slow; therefore, we only have shadows cast for the dog, the ball, the perimeter walls, and the rocks. We would like to have the ground also cast shadows, but since the mesh is not fine, the shadows cast from the hills cannot be seen.

The light is implemented as a combination of ambient light and a spotlight. We used the WebGL renderer with soft shadows enabled. For the shadows, we used a spotlight set from a far distance to give the appearance of sunlight. To give make the area brightly lit we also used an ambient light. The shadow-map size is 2048, which is large enough to encompass the entire map. We had issues with the implementation of the shadows, and the terrain is not shaded correctly. However, we were able to implement shadows for the rocks as well as the walls.

The skybox implementation was taken from the examples [5]. The texture for the skybox was taken from [6].

Movement Logic

The movement implementation for the human player and dog was also taken from the PointerLockControls example. The movement is animated in an animation callback. The animation is updated by taking the time between the update frames and changing the player velocity based on the time. The velocity does not change unless the player is moving. The movement controls are updated via keyboard event listener. There is a bug with the jumping velocity because we implemented a height raycaster. This conflicts with the original PointerLock movement because it depended on the height to remain constant on a flat surface.

For the dog, the movement is similar to the payer movement. However, the notable difference is that the dog pauses the animation when it is not moving, and resumes the animation whenever it is moving.

The ball is switched from the player's view to the scene by removing the object from the player and adding it to the scene and vice versa. This is also allows the player to reset the ball if it is lost or if there is a glitch by using the ENTER key.

Physi.js

Physi.js is a simple and flexible physics plug-in for Three.js. We used Physi.js over alternatives due to its similarities to Three.js when creating objects and rendering [7].

Ground and Physical Objects

For the ground, we implemented a simplex noise generator from [8]. This noise generator allowed us to create a terrain that had hills, but also physical material properties which could interact with the ball. To make the terrain we use 10 vertices, which could be changed into hills. However, there is a bug that occurs when the ball has too much speed and travels through a hill whenever it is rolling. We want to fix this by implementing a finer mesh and finding better intersections.

The ball is a sphere with an applied texture and has a bouncy material. To stop the ball from rolling too much, we implemented a dampening system that divides the ball's velocity as it rolls until it is below a threshold velocity, acting like friction. The dampening only occurs when the height of the ball is below a certain value, therefore it does not slow down in the air. The throwing of the ball gives the ball an initial velocity vector in the direction of the camera. The velocity vector is calculated by casting a ray from the camera's position to the position of the mouse on the screen [2]. In addition, when updating the position of the ball, there is a "dirty position" flag that must be set to change the ball's position [9]. Whenever the ball is taken by the dog we remove it from the scene and add it whenever it is thrown.

The rocks and walls are also physical objects, and the ball can bounce from them. The walls are four boxes that encompass the perimeter of the field. This prevents the ball from escaping. The rocks are convex polygons of 12 points. The texture sources for the ball and the walls are listed in the sources [10], [11].

Blender

We use Blender to rig and animate the dog mesh model [12]. Initially we were considering using animated sprites (GIF images on moving boxes), but this idea was not immersive and the dog mesh model was well-suited for the application. Therefore, we used Blender because it could apply a skeleton to the dog model and let us create a custom animation for movement.

Rigging

Initially we rigged a very simple bone structure to the mesh consisting of 6 bones, and animated the simple skeleton to walk first. After we could animate the dog with a simple skeleton, we rigged a complex skeleton of 30 bones which fully populated the mesh body of the dog. With this rigged skeleton, we could then animate each of the limbs, the head, and the tail of the model with more realistic accuracy. The limbs are broken down into 5 bones each with individual bones corresponding to the paw, the elbow, and the clavicle.

Animation

We followed keyframe breakdown of a running dog, and matched our rigged model to six keyframes and use blender's automatic forward kinematic rendering to match the six keyframes into a run cycle [13]. The keyframing initially was difficult because, in order to make each keyframe we needed to consider the height of the joints and whether or not the forward kinematics would find the right motion. Further, there were difficulties in moving the bones because some bones did not perfectly match the corresponding area to the mesh and resulted in semi-warped mesh on some frames. This is because during our rigging we used Blender's auto weight assignment tool to assign bone weights to mesh regions.

Exporting

We encountered many issues during the exporting of the completed animated dog from Blender to Three. Three has developed their own Blender add-on, but the settings to export must be exactly the same as the documentation, otherwise there is no compatibility. The most difficult part of exporting is the lack of documentation with the exporter, which made it difficult to troubleshoot how to verify the export [12].

Importing

We imported the animated model into Javascript with Three.JSONLoader, which loads animations. The model is a skinned model, which means that the material associated with the model is loaded with the model. Further, the animations were loaded with the THREE.AnimationMixer, which potentially can load multiple animations.

Sound

The background music is from Animal Crossing and the bark effect is from freesounds.org [14][15].

Conclusion

We created a game with custom animation and interactive elements. We used Three.js, Physi.js, and Blender to create the game mechanics and render the game. The logic is intuitive and the result is easy to play with two people.

For future work, we would like to resolve the current bugs and implement some new features such as: a dog oriented view for the dog player, multiple dog choices, more dog animations, a more fleshed-out environment, and a scoring mode.

Sources

[1] https://github.com/mrdoob/three.js/
[2] https://threejs.org/examples/misc_controls_pointerlock.html
[3] http://stemkoski.github.io/Three.js/Viewports-Minimap.html
[4] https://free3d.com/3d-model/tree-72460.html
[5] https://stemkoski.github.io/Three.js/Skybox.html
[6] https://opengameart.org/content/sky-box-sunny-day
[7] https://github.com/chandlerprall/Physijs
[8] https://github.com/chandlerprall/Physijs/blob/master/examples/heightfield.html
[9] https://github.com/chandlerprall/Physijs/wiki/Updating-an-object's-position-&-rotation
[10] https://www.123rf.com/stock-photo/kickball.html

[11] https://www.pinterest.com/pin/213076626097178024/
[12] http://unboring.net/workflows/animation.html
[13] artattaks.canalblog.com
[14] https://www.youtube.com/watch?v=JkHgU_O6f64
[15] freesound.org/people/jorickhoofd/sounds/160094/
[16] Dog Mesh: https://sketchfab.com/models/1c8c763518ab4751bfcddf0b6a34011a