3D Knockout Written Report

Richard Wolf, Jonah Lytle, and Jacob Shachner

Abstract

3D Knockout is a 2-4 player game based off of the 2D Game Pigeon game “Knockout,” in which players launch their penguin characters with specified magnitude and direction in an effort to knock the opposing player’s penguin characters off of the ice sheet that the game is played on. A player begins with four penguin characters and wins by knocking all of the opponents’ penguins off the ice sheet while having at least one of their characters remaining in the end. Each player gets one opportunity to “launch” their penguins per round, and the penguins begin to move according to the user’s instructions simultaneously at the end of the round. In our project, we managed to implement all of the key aspects from the original 2D game into a visually appealing 3D scene containing the icey playing field, an animated ocean, mountains, and several dynamic, exotic creatures. However, we also implemented other cool features such as incorporating a scoreboard, sound effects, a rotatable camera angle, and the ability to play with 2-4 players.

Introduction

The goal of this project was to create an enhanced version of the popular 2D Game Pigeon/iMessage game “Knockout” in an aesthetically pleasing 3D space. The original “Knockout” is a two-player game that takes place on a square sheet of ice and begins with each player having four penuin characters. The objective of the game is to “launch” your penguins in an effort to knock all of your opponent's penguins off of the ice sheet into the ocean while keeping at least one of your penguins on the sheet. Each round both players choose the magnitude and directions in which they want to launch their penguins. The penguins are then launched simultaneously and the resulting movement and collisions occur based on the directions given by the players. In the iMessage game, users launch penguins by placing their finger on a certain penguin and sliding their finger with the desired direction and force. While the iMessage game is great in that it conveniently can be played through iMessage on any iPhone and has a very fun overall concept to the game, it is not particularly stimulating visually and is only playable for two players at once. In our project, we wanted to produce a fun and visually impressive version of “Knockout” that anyone and up to three other friends would have a blast playing together.

In order to create an enhanced “Knockout” game designed for PC web browsers, we tried first and foremost to implement all of the original aspects of the game using javascript and Three.js. Additionally, we wanted to give the option to select how many players were playing and make the game playable for up to 2-4 players. We also wanted to make the game as visually interesting as possible by animating the surrounding water, including realistic looking penguins, adding sound effects, and rendering other interesting objects around the scene.

Our approach to develop the project was to first focus on getting a properly functioning minimalistic version of the game, and then to develop more elaborate features only after this was achieved. In order to accomplish this, we had the plan to initially only load the square ice sheet

and a few penguins into the scene, and then work until the physics of the game were correct. After the correct mechanics were implemented, our approach was to add other integral components of the game, such as enabling user input, creating rounds for the games, multiplayer capabilities, and creating an initial welcome page that allows for the number of players to be selected. Only after all of these key features were implemented, our approach was to make our game as visually stimulating as possible by adding more elaborate features to the scene. Overall this approach worked very well by providing implementation structure during development and ensuring that the most important features were accomplished well before the deliverable deadline.

Methodology

Penguin Physics

The integral physics components of the game that needed to be implemented were the initial penguin launching, the handling of penguin collisions, friction between the ice and penguins, and gravity. To handle the physics, we created a penguin class that stored a number of instance variables such as velocity, netForce, and coordinates that maintained information necessary for physics implementation. The velocity, netForce, and coordinates of a penguin were represented as Vector3 objects in xyz order. In addition, a number of functions were created in the Penguin class such as applyGravity(), collide(penguin), launch(vector), and applyFriction() that when invoked applied a particular force on this penguin. To apply gravity the netForce of the penguin is set to (0, -60, 0). The collide function checked if the penguin specified is in the process of colliding with the penguin provided in the argument. If the two penguins were within two times the radius of the penguin and were moving towards each other, then they were determined to be colliding. In this case, this function updated the velocity of both the penguins to take into account the collision and returned true. If the penguins were not close enough or were moving away from each other (had just collided) then the velocities were not changed and false was returned. To launch a penguin its velocity is set to equal the parameter vector, which is set using the user's specified arrow as we will discuss later in the report. In order to give the appearance of friction to a penguin we set its netForce equal to a vector with the exact opposite direction of its motion with 1.5 times the magnitude of its velocity. We make sure to only apply a frictional force if the penguin is in motion on the ice sheet.

After the penguin class was implemented so that forces were capable of being applied to individual penguins, a number of functions were created in the Scene class in order to apply forces to all penguins in the scene simultaneously. The first two of these functions were handlePenguinsoffIce() and handleFriction(), which iterated through all penguins and invoked the Penguin applyGravity() function if a penguin's position was outside the bounds of the ice sheet and invoked the applyFriction function for each penguin respectively. The next function implemented was the handlePenguinsCollisions() function, which invoked the Penguin collide() function once per each pair of penguins in the scene.

Once the forces were applied when necessary, we implemented the key functions updateVelocities(deltaT) and updatePenguinPositions(deltaT) in order to actually produce penguin movement in the scene. For every timestep deltaT, which we eventually selected to be 0.01, these functions calculate and update the new velocities and positions for each penguin in

the scene. The updated velocity for each timestep was set to equal the velocity in the previous timestep plus the current acceleration times deltaT. Using this new velocity, the new position was calculated to be the previous position plus the updated velocity times deltaT. Doing this for each timestep allowed for continuous penguin movement.

For every update to the scene, handlePenguinsOffIce(), handlePenguinCollisons(), and handleFriction() were first called to update the forces acting on each penguin in the given frame. After the forces were updated, the updateVelocities() and updatePenguinPositions() functions were called to actually move the penguins accordingly. Doing all of this enabled us to fluidly render penguin motion in our game. We believe that this incremental time step approach made the most sense for rendering motion. This approach was the first way of implementing physics that we conceived, and pursued it without ever encountering issues given the size of our project.

Performing Rounds

A majority of the minimum viable product occurs through each time we perform a round. The code for each round is found in the method, performRound(camera) method. The first thing in this method is that it checks if a message needs to be sent to a player indicating that it is their turn to play. This is only determined after the first iteration of performRound is called due to code having to determine which player that the message needs to send the message to.

At this point, I will talk about the event handler method that we implemented to help transition throughout the game. In the handleImpactEvents(event) method, there are if statements which handle different cases for the buttons that are pressed. If the enter key is pressed and there is a message on the screen, the code will ensure that the message is removed and the player who clicked enter will have the option of selecting the direction for their first penguin left on the ice. If there is no message displayed, this would lead us to be during the selection of velocities for the penguins and therefore it will determine if there is a next penguin for the current player to give a velocity to, and if not then it will see if there are any other players who have not yet selected their velocities who have penguins on the board. If any of the arrow keys are pressed and a player is selecting their velocities for a penguin, the arrow will change directions depending on the arrow key pressed. All arrows are created through the drawArrow(currentClick) method, which takes in a position and given the current penguin, found through the current state of the game, it draws a new arrow using the ArrowHelper class. We chose to use the ArrowHelper class due to it being a simple way of drawing an arrow on the screen.

Once the first message is read, and the first player clicks enter, they are then given the ability to determine the speed and direction that they want each penguin to have using the arrow keys. Once they have a direction and speed that they are happy with, they click enter to move to their next penguin and repeat until they no longer have any more penguins left to decide for. A new message pops up to signify the need to change players, all arrows are removed from the screen so that the next player cannot see those arrows, and the cycle repeats until every player who has a penguin left on the ice has selected their direction and velocities for all penguins. Once this happens, the penguins all shoot off with a velocity determined by their players, and experience both collisions and friction to determine where they stop. Once they stop, the ice rescales to shrink to make it harder to stay on the ice. We stopped shrinking at round 7 because we determined that having the ice shrink anymore would gain no benefit, and that making it

smaller would only make it harder to win the game. We then repeat this until there is only one penguin left on the ice, or no penguins remain on the ice.

Displaying Messages

In order to display messages, we decided to write html and insert it into the document. For the start page we reorganized the flow of the app.js program. We created a formatted html start page and then attached event listeners to the keys, '2', '3', and '4' as well as the buttons on the page. Once one of these events occurred we then initialized the scene, camera, renderer and game. For the scoreboard we created html for the format we wanted the scoreboard to take, then wrote update functions, so that the html could be updated whenever the score needed to be updated. We set the css style variable position to be fixed so that the score always remained on the screen. For the popup message we initialized a popup screen and altered the css of that message so it was hidden. We created an update function so that if we called the update function, it displayed the popup with the header and text provided to the function. We also had a remove function which hid the popup when the user dismissed the alert.

Adding Sound

Adding sound to the game was relatively straightforward. We first imported two sound effects that we thought would be good additions to the game: a splash for when the penguins fall into the ocean and a penguin squawk for when the penguins collide with one another. In the scene updatePenguinPositions() function, I check if the penguins position is below the surface of the water in the scene. If so, I create an audioLoader, load the imported splash audio, and play the audio. In the handlePenguinCollisions() function, I check whether or not a pair of penguins collided into one another. If so, I perform the same process as before to load and play the audio of a penguin squawk. Doing this causes a new audio snippet to be generated every occurance of a collision or a penguin reaching the surface of the water. Other implementations could have also been to include music or ocean noises in the background as well, but we found that overlapping too many different audio samples at once led to unpleasant results. As a result, we opted to just include sound effects. The sound effects in our final product are pleasant for the most part, but can sound off if too many collisions occur at once.

Importing Objects & Enhancing Scene

In order to make our scene as visually appealing as possible we imported a number of objects and animals (both animated and still) to create a whole environment around the game. Our imported objects came from Google Poly, Three.js, free3d, sketchfab, and one independent modeller. Depending on the 3d model type, we had several different processes for importing and adding the model to the scene. We used the Three.js OBJLoader class to load .obj files along with MTLLoaders if we needed to load a material or texture. We also used the GLTFLoader to load .gltf and .glb files, once again using MTLLoaders to load the materials and textures. To enhance the scene further, we animated a number of the objects, notably the water, shark, flamingo, stork, parrot, and mosasaur. For each of the animals we used sine and cosine functions to make the movements seem realistic and for the water we used the animation as provided in the Three.js source code.

Results

Our end goal was to create a fun and visually stimulating version of "Knockout" that users would enjoy playing with 1-3 friends. Therefore, we thought the best way to measure the success of our project was to have users play our game and then to answer a few questions based on their experience. These questions consisted of "Rate how much you enjoyed the game on a scale from 1 to 10," "Rate how aesthetically pleasing you found the game on a scale from 1 to 10," and "please provide any other additional feedback."

After 20 people played our game and provided feedback, we found the average enjoyment rating to be 9.1 and the average aesthetic rating to be 9.3. As these averages are both above 9/10, we concluded that people really enjoyed playing our game and thought it was visually attractive. Consequently, we considered our game to be successful based on these metrics. The additional feedback we received was overwhelmingly positive and included comments such as "This game is better than the original!" and "It is very pretty! I really enjoyed just looking around at the scene at times." These comments further indicated that people thoroughly enjoy both playing and viewing our game.

Discussions

The approach that we took as far as we can tell has been very promising. Our final product exceeded all expectations that we had, and throughout the process we all learned a lot not only about how to effectively create a game, but also what it takes to work in a group to complete a project as complex as ours. The approach was to first get our game to a minimum viable product as quickly as we could, while modularizing all aspects along the way. This way, once we knew that the basic functions worked, we could go ahead and polish the game to make it appear and play as a complete game. We also broke down the project into tasks and the three of us took leads on the different areas to ensure that they were completed and worked with all other components of the game.

The direction that should follow our game should be focused on the multiplayer aspect. What we mean by this is adding the capability to play the game on different devices. We knew that this was something that would most likely not get done during our short time frame of working on the project, and had to "settle" (we do not think that our game settled in this aspect and feel that it makes for a more enjoyable game due to seeing reactions in real time of the other players) for taking turns on the same computer. We feel that this works due to clear instructions highlighting when to move to the next player. We also have played plenty of multiplayer games on the same computer to know that it can still be an amazing experience for the user to take turns instead of being on seperate computers, and we hope that you feel the same experience we feel.

To build on what we learned, group programming is definitely something that we all improved on, especially not being able to physically be in the same location or area as each other. This really tested our communication and our code commenting abilities as we couldn't waste time trying to figure out what to do next or what certain code meant. Another thing that should not be overlooked is how much we learned about ThreeJS and JavaScript. We all feel so much more prepared if we ever encounter either in our future endeavors.

Conclusions

We believe that we achieved our goal successfully. Not only did we convert the original game from 2D to 3D, we made the scene much more visibly appealing and interesting, as well as incorporated new features, playing with 2, 3, or 4 players as opposed to just 2. Additionally, the Results section of the report makes evident that we also accomplished our goal of making a game that people thoroughly enjoy viewing and playing. For future steps we would like to include player customizations for the game. For example, we would like to give players the ability to change the weather, scene variables, player names, and player type (other animals rather than penguins).

Contributions

Richard - Contributed to physics oriented code in both Penguin and Scene class. Helped create the rounds functionality. Contributed to the motion of background animals in the scene. Created an introductory camera sweep of the scene at the beginning of the game. Added sound to the game.

Jonah - Helped import and create objects. Animated objects and helped code the motion of the animals in the background. Contributed to physics code in Penguin and Scene. Wrote HTML for scoreboard, welcome page, and popups. Color coded everything to match each player. Changed flow of app.js to allow for the start page and added key listeners to streamline the starting process.

Jacob - Contributed to implementing the game rounds. Through this implemented the user interaction for selecting the direction and velocity of the penguins, and determine how to both draw an arrow for the user to see and convert that into a starting velocity. Helped with testing all features.

Works Cited

Flamingo, Parrot, Stork Credit: Mirada: https://mirada.com/ from Rome: http://ro.me/
Implementation help from:
https://github.com/mrdoob/three.js/blob/master/examples/webgl_lights_hemisphere.html

Shark Credit: Poly by Google (https://poly.google.com/view/1mVWW4RFVHc)

Penguin Credit: printable_models (https://free3d.com/3d-model/penguin-v2--128210.html)

Mountain Credit: bilgehan.korkmaz (https://sketchfab.com/3d-models/praise-the-sun-737e12518b1b4e3da8e4406d7da83b90)

Mosasaur Credit: Hoai Nguyen (https://poly.google.com/view/6S_UehMUbiW)

Island Credit: VR XRTIST (https://poly.google.com/view/dBGGbCMGTRu)

Water Credit: Three.js with help from https://github.com/mrdoob/three.js/blob/master/examples/webgl_shaders_ocean.html

Ice Credit: Three.js

Squawk Sound Effect: https://retired.sounddogs.com/sound-effects/birds-arctic-penguin-single-381107

Splash Sound Effect: https://bigsoundbank.com/detail-1519-splash-big-1.html