CS175 Final Project: Pool

by Changseob Lim and Sam Bieler

Usage Instructions

Run Game

In order to run the game you can run the 8BallPool.exe file inside of the 8BallPoolWindows+(your sys architecture) folder if you are on Windows. If you are on mac simply run the 8BallPool Application file in the main directory. As our computers are Macs this is the only build that has been tested. You can also run the game by opening the project in Unity and pressing the play button.

Controls

- Press and hold Spacebar key to take a shot when the pool cue is on the screen.
- Press "j" key to shift the ball left after a scratch or at the start of the game.
- Press "l" key to shift the ball left after a scratch or at the start of the game.
- Press left arrow key to angle the pool cue to the left from the perspective of the camera when lining up a shot.
- Press left arrow key to angle the pool cue to the left from the perspective of the camera when lining up a shot.
- Press left shift key to switch the camera view between bird's eye and cue view except when balls are moving.
- Press "b" key to change between day and night mode.

Overview

As our final project, we created a game of pool using Unity. The following are the five main components that went into our implementation of English pool.

- Learning the Basics of Unity
- Scene Creation
- Game Mechanic
- Camera Control
- Light Control

Learning the Basics of Unity

We started our project by getting familiar with Unity's interface. Being absolute beginners, we approached the project by first researching rudimentary Unity projects that others have previously completed. Our idea to create billiards first came when we were fooling around with one of Unity's publically available introduction projects called Roll-a-ball.

Roll-a-ball is a "game" that has a ball that the player can control and 12 pick-ups that gain the player points. Doing a deep dive into the project files not only gave us the inspiration for pool, but also taught us some important aspects of Unity that we used as foundation for the rest of our project.

We learned that Unity's UI was an important resource to not rewrite code that has already been completed by the creators of Unity. Object creation, scene initialization, and simple attribute allocation were all available, and we did not have to write a single line of code to do these things. Using new visual materials, physics materials, and creating prefabs or templates for assets were also equally straight-forward, although we did have to create some on our own from scratch.

Roll-a-ball was also an incredible introduction to scripting (including collision detection and transformation), and its camera and player scripts were very helpful for our understanding of how to manipulate different attributes given to a specific object and how Unity processes information from its objects in each frame through Update() and in each physics engine update through FixedUpdate(). It also gave us a basic example of how to check for object collisions and run code when these collisions occurred using the OnCollisionEnter function.

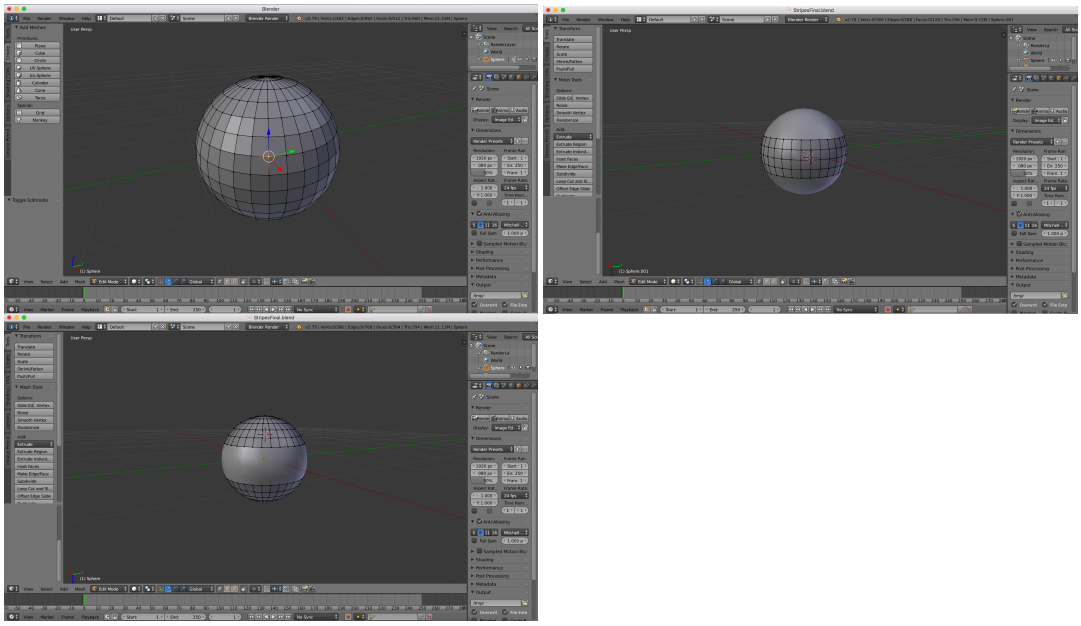
Any other functions or features used were found in the documentation and the manual for Unity 5.x, and they will be mentioned throughout this write up.

Scene Creation

The scene was created using readily available resources provided by Unity. Our scene is composed of the following GameObjects (provided in the parentheses are Unity's basic 3D object shapes that were used to make the object):

- Cue ball (Sphere)
- Cue stick (Cylinder)
- 7 solid balls (Sphere)
- 7 striped balls (Blender Mesh)
- 8 ball (Sphere)
- Table (Plane)
- 6 walls (Cube)
- Kill plane (Plane)
- Catch plane (Plane)

Building Meshes



Out of the GameObjects listed above, only the 7 striped balls are not made up of a basic Unity shape. Vanilla Unity only supports creating materials that are solid in color, so it is rather silly to differentiate them as "solids" and "stripes." At first we could not figure out how to solve this issue, but eventually learned that you could create assets that had multiple submeshes, and each of these submeshes could be colored separately. We wanted to make our game be as close an imitation of the actual pool as possible, so we learned the basics of a software called Blender in order to build meshes for our striped balls. Above is a before and after of the sphere asset that we created in blender. The first image is a photo of the original vertices of a sphere object. The next two photos show the vertices of each of the two submeshes that we created in order to color the striped balls in two different colors. This was only a simple use of the Blender application, but it certainly helped give us a sense of how assets would be created.

Game Mechanic

Logic

All of game logic is implemented in CueStickController.cs

As far as win conditions and rules of the game go, we tried to emulate the real-life pool game as much as possible. Currently, our logic is as follows:

- Win Conditions:
    - A player wins if he sinks all of his balls and then sinks the 8 ball
    - A player loses if he sinks the 8 ball before sinking all of his balls
- Ball Assignment Condition:
    - The first player to sink a ball is assigned the type of ball that was sunk, and the opposite is assigned to the opposing player
- Turn Change Conditions:
    - Turn does not change if a player sinks a ball of his type

- Turn does not change until all balls stop moving
- Turn changes if a player only sinks balls of his opponent's type
- Turn changes if a player does not sink any ball
- Turn changes if a player scratches the cue ball (hits the cue ball out of play)

Cue Stick

The cue stick acts as an indicator for the players to know how the balls are going to be hit, and as an indicator for when you are able to hit the ball. It can rotate around the cue ball using the arrow keys, and this rotation changes the angle at which the force will be applied to the ball. In order to calculate how to apply the force we simply calculated the vector from the pool cue to the cue ball, set its Z value to zero in order to avoid bouncing and then normalized it and finally multiplied it by a constant and the amount of time the space key was held down. We decided to not give the cue stick a collision component in order to make sure it does not interact with neighboring balls or walls.

The position of the cue stick is always set to be an offset from the position of the cue ball. This means that the cue stick will always be pointing in the same direction when it is time for one of the players to shoot. Originally, we encountered an error where at various times the cue stick was not located in the correct spot after a shot. We found out that this was due to the change of orientation of the cue ball, since it sometimes rotates during gameplay. We solved this issue by resetting the cue ball's rotation every shot. This does not change the behavior of the game, as the orientation of the ball does not matter when the balls are stationary.

Registering Balls Made

In order to determine whether or not a ball has been sunk into a pocket we created a plane beneath the table through which the balls must pass. This plane acts as an event trigger. If any ball crosses it, the game increments the score of whoever's ball that was by checking if its tag is "Stripes" or "Solids" or "8 Ball". As mentioned above, the game ends either way if the 8 ball hits the scoring plane.

This implementation worked perfectly until we decided to change turn change mechanic from waiting until the cue ball stops moving to waiting until all balls stop moving. Since the scoring plane does not actually despawn the balls, the sunk balls were at freefall until the reset of the game, and in turn, did not allow for the change of turns. We fixed this by adding another plane of the same size just underneath the scoring plane as the catch plane to catch all fallen balls. The catch plane's friction and bounciness coefficients are at their max values, so the balls quickly come to a halt the moment they hit the plane, thus allowing for the game to continue.

Start of Game and Scratch Behavior

Currently, the state of scratch only occurs if the cue ball goes out of play (hits the scoring plane). When a player scratches the cue ball, the turn changes to the other player, and the cue ball and cue stick are reset to the game start position. In this state, the player is allowed to move the cue ball side to side to choose where to shoot it from.

Wall Glitch

In our original implementation, balls would sometimes escape the boundaries set by the wall if going at a fast enough speed. We realized that this was happening due to the physics engine checking the scene too infrequently. We solved this bug by increasing the speed at which the engine runs.

Camera Control

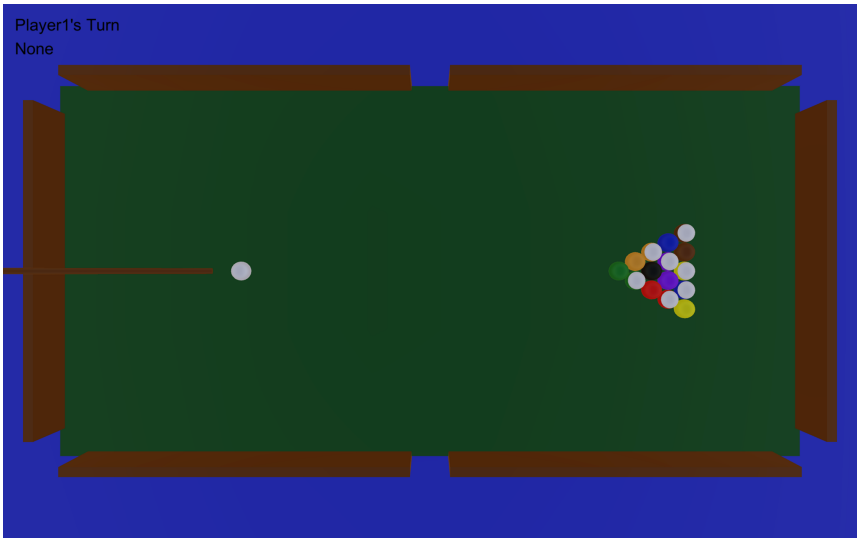
All of camera's transformations are implemented in CameraController.cs.

The camera has two modes: one that follows the cue stick, and another that provides a birds eye view of the entire table. At the start of the game and before each shot, the camera is defaulted to cue stick view. At any time when no balls are moving, the user can click their 'LeftShift' key to toggle between cue view and birds eye view. However, when the balls are in play, the camera is locked to the bird's eye view, and the user has no control over the camera or the scene.

Toggling

Toggling between the two modes is achieved by getting the 'LeftShift' press through Input.GetKeyDown() and keeping a boolean to store which view is currently being used in the scene. We originally toggled between the modes in FixedUpdate() that gets updated at each physics step. However, while fixing the wall glitch mentioned above, we found that this resulted in frequent double pressing of the 'LeftShift' key. We realized our mistake, and moved the toggle to Update(), that only gets updated at each frame.

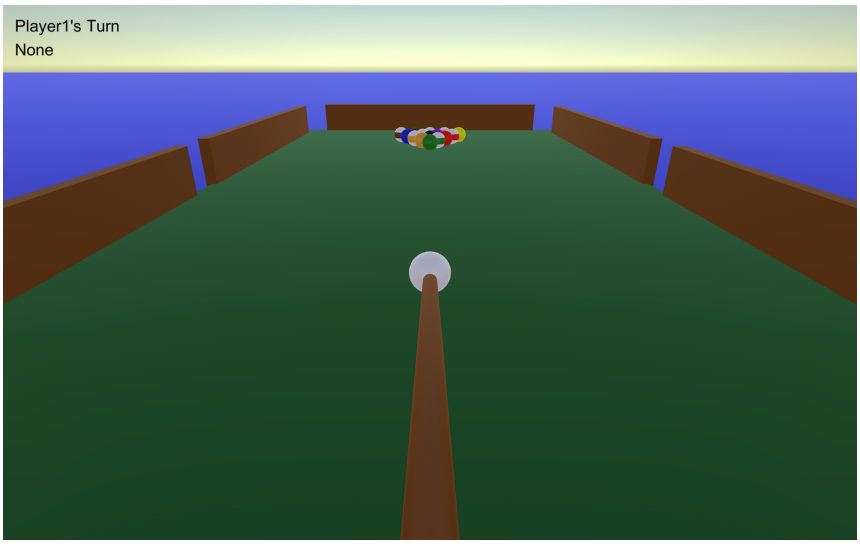
Bird's Eye View



Changing the camera to the bird's eye view was simple. Since Update() is called at every frame, we simply check if any of the balls are moving. If there is, then the camera uses Vector3.Lerp() and Quaternion.Lerp() functions to smoothly transfrom itself from its position to a fixed position over the table. A translation.y value of 25 was used because it snugly fit the entire table to the screen, and both rotation.x and rotation.y values of 90 degrees was used to orient the camera in the right direction. Additionally, these same interpolations are used when the player switches to the bird's

eye view manually.

Cue Stick View

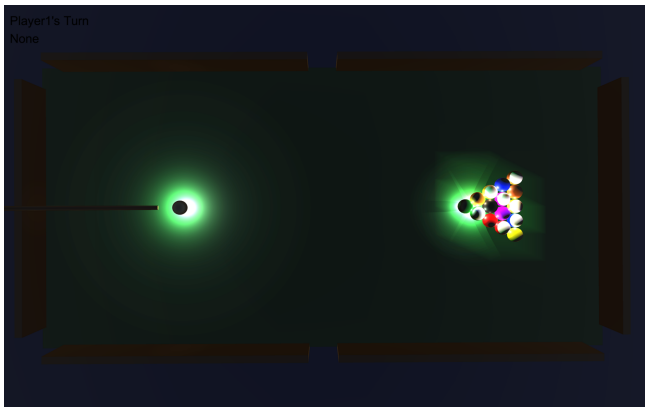


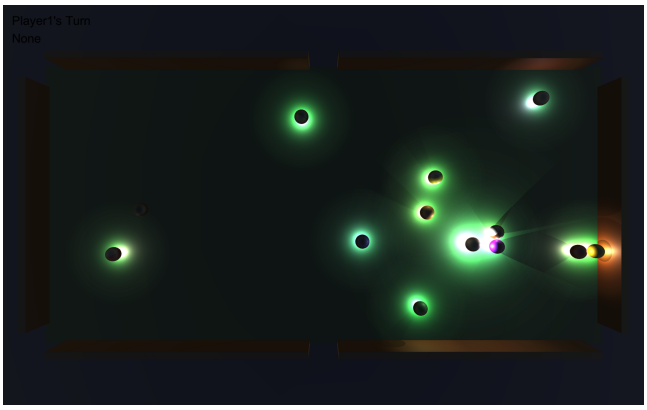
Implementing cue stick view was a little bit more complicated. At first, we used transform.RotateAround() function to mimic the behavior of a camera locked on and rotating around the cue ball. However, this approach was ultimately unsatisfactory, as it was incredibly convoluted to find the correct position and rotation of the camera between each toggle between the two views.

Ultimately, we decided to move the camera in cue stick view using the cue stick frame, so that we can simply translate the camera to the same coordinates every time. This way, the translation is simple and elegant, and we can easily take care of the camera's rotation using Quaternion.LookRotation() function to "lock" it onto the cue ball. The conversion to cue stick frame was completed using transform.TransformPoint() and transform.InversTransformPoint().

Because we are using Lerp() to do all of our camera transformations, everything is animated, even when it follows the cue stick on button press. Although we had the option to not use Lerp() in this instance and make the camera seem glued onto the cue stick, we thought that the effect of the camera "following" the cue stick was both aesthetically pleasing and visually intuitive.

Light Control





After being familiar with Unity's Lerp functions, light control was pretty easy to implement. In Unity, light primarily has two components: color and intensity. For the ambient light in the scene, we decided to keep its intensity constant and Lerp the color of the light from a blueish white to near-black. On the 'b' press at the start of the game, the effect is a slow animation from a bright to a nearly invisible scene - so we decided to add more sources of light as it got darker.

In Unity, every object can act as a source of light by adding a light attribute to it in the inspector. We made every ball other than the 8 ball a source of light, in order to give the nighttime table an arcade air-hockey look. This time around, we lerped the intensity value for the lights in the balls, and made it get brighter in the same rate that the room gets darker.

The lights and shadows of our first implementation looked very polygonal and pixelated. We later found out that the problem was a low count of pixel light count, which is a project variable that we had not accounted for. After we increased the value, the night view of our pool game looks true to our inspiration. The left image above shows the board before the pixel light count value change, and the image on the right shows the board after the change.