CMSC 425
Programming Assignment 1, Part 1 Implementation Notes

Disclaimer

We provide these notes to help in the design of your project. There is no requirement that you use our settings, and in fact your encouraged to be creative and experiment. These notes have not been carefully proofread, and we apologize for any errors. If there is any other information that would be useful, please let us know.

Environment

The bowling-lane consists of the following parts:

- Bowling lane: A 3-D rectangle scaled 10 x 1 x 50 and positioned at (0, -0.5, 0). It has a dark blue color RGB = (30, 50, 150).
- Base: The lane is floating in space, and it sits above a green base. This is a 3-D rectangle scaled to 16 x 1 x 60 and positioned at (0, -2.5, 0). Its color is RGB = (80, 255, 64).
- Walls: The walls enclosing the lane are made up of four 3-D rectangles. The left and right walls are scaled to 2 x 5 x 60 and are positioned at (±9, -1.5, 0). The front and back walls are scaled to 20 x 5 x 2 and are positioned at (0, -1.5, ±31). They are colored yellow, RGB = (255, 255, 64).

Camera

The camera follows the bowling ball. Its initial position is at (0, 7, -30) and it has been tilted slightly downwards rotated with the Euler angles (24.5, 0, 0).

We replaced the default Unity skybox with a solid background color. This is done by setting the Camera's "Clear Flags" parameter to "Solid Color" and we set the color to an orangish shade, RGB = (200, 160, 100).

As in Roll-A-Ball, the camera follows the ball, but it stops moving once the ball moves beyond the front bowling pin. (You will need to modify the CameraController script to implement this behavior.)

Lights

We used a single directional light with the Euler-angle rotation (50, 60, 0).

Bowling Ball

The bowling ball is a black sphere. It is scaled to 2 x 2 x 2 and its initial position is at (-0.2, 1, -20). We moved it slightly off center in the lane just to break the symmetry when it is rolled straight ahead. To

obtain a shiny appearance, we set the material's "Smoothness" setting to 0.75. The bowling ball is associated with a RigidBody with the default settings (mass = 1, drag = 0, angular drag = 0.05).

The ball's motion is the same as in the Roll-A-Ball tutorial, but we increased the "Speed" parameter to 20.

Bowling Pins

Each bowling pin consists of 7 Unity primitives. We modeled each pin as a prefab, positioned at the center of the lane. All except the eyes are colored using Unity's Default Material.

- Central spine: A cylinder scaled to 0.3 x 1.5 x 0.3, positioned at (0, 1.5, 0). We used the default capsule collider.
- Head: A sphere scaled to 0.6 x 0.6 x 0.6, positioned at (0, 3, 0). We used the default sphere collider.
- Body: An elongated sphere scaled to 1 x 2 x 1, positioned at (0, 1, 0). We replaced the sphere collider with a capsule collider, which we reshaped by hand to approximate the elongated sphere. (The axis is aligned with the y-axis.)
- Ears: Each ear is a Unity capsule scaled to 0.2 x 0.3 x 0.2. The right ear is positioned at (-0.15, 3.4, 0) and is rotated with Euler angles (0, 0, 14). The left ear is symmetrical.
- Eyes: Each eye is a Unity sphere scaled to 0.1 x 0.1 x 0.1. The right eye is positioned at (-0.1, 3.1, -0.25). The left eye is symmetrical.

These components are made the child of an empty Unity object, which is associated with a RigidBody. We just used the default settings (mass = 1, drag = 0, angular drag = 0.05). Unlike the pickups in the Roll-A-Ball tutorial, we did NOT set the "isKinematic" parameter, and we do "Use Gravity".

In our implementation, the pins were placed by hand, like what was done in the Roll-A-Ball tutorial. (A better solution would be to write a script to position the pins.) We created an empty Unity object into which to place the pins, and this object was positioned to align with the frontmost pin at (0, 0, 20). Each of the pins were then placed relative to this (so the frontmost pin has position (0, 0, 0). The rows of pins are separated by the distance of 1.5 units and the distance between consecutive pins in each rot is 1.5 units.

How do you determine when a pin has toppled over? I will let you think about this. Our solution was based on accessing the `transform.eulerAngles` component and analyzing how far the object has rotated. A rotation beyond 45 degrees away from vertical is a good heuristic.

Physic Material

We wanted the physics to be fairly lively, so we associated all the colliders (pins, ball, walls, alley, etc.) with a Physic Material having a "Bounciness" parameter set to 0.5. We left the "Friction" parameters at the default setting of 0.6.

Canvas

We have three text objects in our canvas. One in the upper left for counting the number of pins that have toppled, one in the upper right counting down the remaining time, and one in the center for the

win/loss messages. All the font sizes were set to 24. The upper left text is upper-left justified and was offset by (10, -10, 0), the upper right text is upper-right justified and offset by (-10, -10, 0), and the central text was centered (both horizontally and vertically) and offset by (0, -40, 0). Note that you will need to make the height and width of the Text objects' Rect Transform large enough so that all the text is displayed. For example, we needed to resize our countdown timer to 160 x 30 to get the text to fit. You will need to experiment based on the text and font sizes you use.

We used black fonts for all the text in the upper corners. (In an earlier version, the central text was orange, but I think that black is more legible.)

Scripts

In the Roll-A-Ball tutorial, it is the PlayerController that handles the counting of Pickups. This seems a little less natural for a bowling game, and so we created a class called AlleyController that keeps track of everything (counting pins, maintaining the countdown timer, displaying messages).

Countdown Timer

The 10-second countdown timer was adapted from a solution on the internet. I think that the best approach is to use a coroutine, that wakes itself up every 0.1 seconds and adjusts the time. I found a solution based on that approach here:

https://answers.unity.com/questions/225213/c-countdown-timer.html

(I adapted the second solution posted by U_Ku_Shu. That solution is based on counting in 1-second intervals, but the change to 0.1 was straightforward.)

Another issue is generating an output with the proper precision (e.g., to get 2.7 rather than 2.69999). I found a solution here:

https://stackoverflow.com/questions/33582111/formatting-a-number-using-tostring

Any time you find code snippets like these on the internet, you can use them, but you must cite them in your ReadMe file when you submit your final assignment.

Restarting and Quitting

We have not discussed how to restart or quit a game. The relevant Unity commands are:

```
UnityEngine.SceneManagement.SceneManager.LoadScene(0); // restart the scene
```

and

```
Application.Quit(); // quit the program
```

Finally, although we did not require it, you might be curious how to implement a pause/unpause feature in your game. The following is quick-and-dirty (but not always perfect) solution:

```
Time.timeScale = 1 - Time.timeScale; // pause/unpause game
```