CMSC 425 – Prog 1, Part II
Parameter Settings

This document describes several parameters and settings used in our implementation of Programming Assignment 1 (Tilt Maze), Part II. **You are encouraged to experiment with your own settings**, but if you wanted to imitate our look and behavior, you can use these. (You do not need to acknowledge the fact that you have borrowed these in your ReadMe file, but if you use any other external sources, you must do so.) Note that I have entered these by hand by transcribing them from my Unity project. If something appears amiss, it might just be a transcription error. Please check with me. Changes to existing parameters from Part I are highlighted in red.

**Main Camera**
Our camera is placed at \((0, 9, -4)\) with rotation \((70, -3, 0)\)

**Light Source**
In computer graphics, a light is said to be *directional* if it behaves as if it is infinitely far from the scene. (For example, think of the sun.) Directional lights are more efficient to process, because the light rays are parallel to one another, which simplifies shadow and shading computations. In Unity, the “position” of a direction light source is ignored. The direction is determined by the rotation.

We used the directional light source provided as the Unity default. It has the rotation vector \((50, -30, 0)\), which places the light source above, right, and behind the camera. As done in some Unity tutorials, I moved the light source up to \((0, 100, 0)\) so it is out of sight in the Scene view. The light’s color is given by the RGBA hex value FFF4D6FF.

**The Ball**
Our ball is a Unity sphere of radius 0.5. The initial position is \((0, 3, 0)\). It’s color (albedo) is dark gray (with the RGBA hex value of A2A2A3FF). To give it a metallic look we set the Metallic parameter to 0.9.

You can consult the Unity manual/tutorials for associating materials with objects. Basically, you need to new material in the Project folder and then drag the material on top of the object in the Unity scene view.

To cause the ball to roll freely but bounce off the walls, we created a Unity physic material that is associated with all the solid objects of the maze (the ball, platform, and walls). It is associated with the colliders of these objects. The physic material has Dynamic and Static Friction setting set to zero, and bounciness set to 0.5. Both Friction-Combine and Bounce-Combine are set to “Average”. See the Unity manual/tutorials for information on how to create physic materials and associate them with objects.
When the user hits the space bar, the ball jumps. To do this we applied an impulse force upwards of 4 units.

When the ball hits the goal (after collecting the required number of pick-ups), it floats up at a speed of 10 units per second. (You will need to use a script to cause this effect. Remember to first make the object kinetic, to remove it from control of the Unity physics engine, and then you are modify its transform directly.)

**Base Platform**

Our base platform (on which the ball rolls) is a Unity cube, centered at the origin and scaled to (10, 0.1, 10). This means that it extends from -5 to +5 along each of the x- and z-axes. The material is orange (with the RGBA value C98323FF).

The platform’s collider is associated with the same physic material as the ball.

**Outer Walls**

The four outer walls are Unity cubes. Each is centered at one of the middle edges of the platform. (For example, the west wall is centered at (-5, 0, 0).) Each is of height 1 and width 0.25. The length is 10.25. (The additional 0.25 in length is so that they match up nicely at the corners. If you set the length to 10, you will see that the corners of the maze don’t look good.) The material is a light purple (with the RGBA value A4A3E3FF).

The walls’ colliders are associated with the same physic material as the ball.

**Inner Walls**

The inner walls are placed randomly as described in the project handout. Imagine the 10 x 10 board being subdivided into a grid, with grid vertices placed at integer points lying in the range from 0 to 10. Wall hinges are placed at the even vertices of the grid (vertices [i, j] where 1 ≤ i,j ≤ 9 such that i + j is an even number), but not at the center of the grid (that is, not at [5, 5]). Each wall is of height 1 and of width 0.25. It is centered on the ground (the (x,z)-coordinate plane). The hinge is positioned 0.4 units away from its center. The walls’ colors are the same as the outer walls, and their colliders are associated with the same physic material as the ball.

The initial rotation of each wall is randomly chosen from North, South, East, and West. (Note that inner walls may collide both with one another and with the outer walls.) At random times, but at intervals of roughly 10 seconds, each wall starts to rotate. Once started, it performs a rotation over a period of 1 second through 90 degrees either clockwise or counterclockwise (chosen at random).

**Goal**

The goal consists of two parts, both centered at (4.5, 0, 4.5). There is a trigger collider associated with it to determine whether the ball has hit it. (Beware: Depending on exactly where you place the trigger, it may fire at the start of the program because the platform intersects the trigger. Of course, this event should be ignored.)
One is a green cylindrical base. Its radius is 0.5, and its color has the RGBA hex value 48DD3FFF.

The other is a particle system. To emit the particles upwards, I applied a rotation so the z-axis points upwards. (I’ll let you figure this out.) Compared to the Unity defaults, here are the parameters I used:

- Start Lifetime: 10 (big enough that the particles don’t disappear before they float out of the window)
- Start Speed: 1 (so they appear to float, rather than shoot, up)
- Start Size: 0.5 (artistic choice)
- Start Color: Hex value 89D490FF (to roughly match the cylindrical base)
- Simulation Speed: 0.5 (artistic choice)

I believe that I used the defaults for all the other parameters.

Remark: When the board tilts, the origin of the particle system should move with the board, but the direction at which the particles are emitted should remain fixed.

**Tilting**

Let me remark that I found the mechanism for tilting presented in the Youtube tutorial to be rather inelegant. You are welcome to use it, but let me explain my alterations to the method in the tutorial.

First, I created a public float variable called `tiltSensitivity`, which I set to 20.0f. (Since it is public, the parameter can be adjusted in the Unity editor to achieve the desired degree of sensitivity.)

Second, I didn’t like the fact that the amount of rotation was insensitive to the update time. A program running at twice the frame rate will tilt twice as fast, which is not desirable. So, my tilt angle was scaled by `Time.deltaTime`. For example, my front-back rotation angle was set to:

\[
\text{Time.deltaTime} \times \text{tiltSensitivity} \times \text{Input.GetAxis("Vertical")}
\]

Third, I found his code for limiting the rotation to be a bit clunky. At issue is the fact that Unity angles range from 0 to 360, and so limiting the angle by 10 degrees means that it should be smaller than 10 and larger than 350, which (as he points out in the tutorial) very confusing. My approach was to first normalize the Euler angle values from the range \([0, 360]\) to the range \([-180, +180]\). To do this, I wrote a small function that takes the vector of Euler angles, and for each component that is greater than 180, it subtracts 180. After this normalization, the tilt test is much simpler, namely, the rotation can range from \(-T\) to \(+T\), where \(A\) is the maximum tilt angle. (By the way, remember that the default parameter passing mechanism is by value. If you change the function argument, you want to pass it by reference. Please see the Microsoft documentation on how to do this.) In my implementation, I set \(T = 20\) degrees.

**Pick-ups**

When the game starts up, we generated 6 random pick-up objects. The objects are essentially of the same shape as those in the Unity Roll-A-Ball demo. Ours were Unity cubes, scaled to 0.25, rotated by
(45, 45, 45) and placed 0.25 units above the ground. Each is colored with a slightly transparent (by setting the Rendering Mode to Transparent), with a material that has color (albedo) with the RGBA hex value of F2005A8C. Each pick-up object rotates with the rotation vector (15, 30, 35), as in the Unity demo.

In contrast to the swinging walls, whose hinges correspond with the vertices of the grid, the pick-ups are placed at the centers of the grid squares. Pick-ups are not be placed at the center nor on the goal position.

Note that the swinging walls and pick-ups do not generate collisions. A wall can swing through a pick-up without any effect.

To win the game, the user must gather 2 pickups before moving to the goal.

**Text**

When the game starts up, in the upper left corner we display the text, “Pick-ups remaining: 2”. This counts down until the desired number of pick-ups have been gathered. When the number remaining reaches zero, the text changes to, “Done! Go to the goal!”

On reaching the goal (after acquiring the required number of pick-ups) this text goes blank, and in the upper right we display the text “Congratulations. You win!” If the player jumps over the edge of the board and the y-coordinate of the ball falls below -10, the upper right text reads “Sorry. You lose!”