Programming Assignment 1: Unity Tilt-Maze Game

Handed out: Thu, Feb 2. Due: Part I is due Wed, Feb 8, 11:59:59pm and Part II is due Mon, Feb 20 at the same time. Note the new due date!

Late policy: Up to 6 hours late: 5% of the total; up to 24 hours late: 10%, and then 20% for each additional 24 hours. Submission instructions will be given later.

Overview: The goal of this assignment is to learn the basics of Unity by implementing a classical wooden game called a Tilt Maze, where the user tilts a board in order to roll a marble through a maze.

Part-I Requirements: (Due Feb 8.)

General Structure: The basic structure of the tilt-maze board should be similar to that of the Youtube tutorial. The placement height and width of the internal walls need not be the same as in the introduction. You should at least four walls, some of which connect to each other.

Materials: Associate colors with all the objects of your scene. The ball, platform base, and walls should all have different colors. For simplicity, you can use solid colors (unlike the tutorial, you may use a solid color rather than a texture for the floor of the maze).

As in the tutorial, the walls should be associated with a physics material that results in the ball bouncing off on contact. Keep the friction very low (or even zero) so that the ball rolls easily.

Ball: You may color the ball however you like, but (unlike the walls) it should have a shiny metallic or glassy appearance.

Goal: As with the tutorial, the goal should be represented using a particle system. (You may configure the appearance of the particle system however it suits you.) In addition, there should be a “spot” on the ground that indicates where the goal is located. I represented my goal location as a very flat green cylinder.

Limited Tilt: As in the tutorial, there should be limits on how much tilt is allowed for the board.
Start and End: We will make the start and end states more complex for Part II, so you can keep it simple for now. The ball should start floating above the starting point on the board, and then fall onto the board when the game begins. (I just used gravity to do this.) When the ball reaches the goal state, it should rise up and out of the image to indicate that the game is over.

Quit/Restart: It should be possible to quit the game at any time (e.g., by hitting the ‘Q’ key). It should also be possible to restart it to the original state at any time (e.g., by hitting the ‘R’ key). Note that when you are running the game within the Unity editor or on the web, you may not be able to quit the game. (Restart should work, however.)

You are allowed to modify these specifications (e.g., by altering the models, colors, and some aspects of game behavior), provided that your game demonstrates that you have mastered all the required elements listed above. For example, if your game uses mouse input rather than keyboard input, your player object should still be based on physics forces (as it is in the demo), and you should have some form of keyboard input (to demonstrate that you know how to do this).

- The game can be restarted by reloading the initial scene. This can be achieved by invoking `UnityEngine.SceneManagement.SceneManager.LoadScene(0)`.
- The game can be quit with the command `Application.Quit()`.
  Note that quitting the game while it is running within the Unity editor or a Web-based deployment does not do anything. (At least, it didn’t do anything with my implementation.) However, if you produce an stand-alone executable (e.g., an “.exe” file on Windows) then quitting the game will terminate the program.
- In order to get control of the ball at the end of the game, get the ball’s `rigidbody` component and remove it from control of the physics system by setting `rigidbody.isKinematic = true`. This puts the player under the full control of the program, and you design a script to control its movement.
- When you restart the game, does the lighting suddenly change (becoming much darker)? This is a common issue. For help, see the following post on answers.unity3d.com.

Part-II Requirements: (Due Feb 20.) This is mostly an extension to Part I, but there are a few modifications as well.

Ball: The ball is essentially the same as in Part I, but we reduced its radius from 0.8 to 0.5. The ball should now have the ability to “jump.” When the user hits the space bar, the ball jumps vertically. (This can be done by using the function `AddForce` to apply a vertical impulse. Note that the direction should be independent of the board’s current tilt.) The size of the force should be large enough so that the ball can hop over a wall. Try not to make it too high, since otherwise it is hard to control where it lands. If the ball hops off the edge then the game is lost. (Indeed, this is the only way to lose in our implementation.)

Walls: Rather than placing the internal walls in the editor, the walls will be generated by a script. The generation process is random, so that each time you restart the game, you will have a new set of walls.

Each wall is roughly the length of a cell of the grid and behaves like a swinging gate. The hinges about which the walls spins are fixed, but their orientations are random. Think of the platform as decomposed into a $10 \times 10$ grid (see Fig. 1). The grid vertices are numbered from 0 to 10 along each side (with 0 and 10 corresponding to the outer walls). The hinges of these swinging gates are placed at the even vertices, that is, the indices $(i, j)$ such that $1 \leq i, j \leq 9$ and $(i + j) \equiv 0 \pmod{2}$. (These are indicated with small black circles in Fig. 1). The only exception is that there is no wall placed at the center point, which indices $(5, 5)$, since this is where the ball drops.
Initially, the orientations of the walls are random, pointing either north, east, south, or west. (You may find the Unity function Random.Range(0,4) useful. It generates a random integer in \(\{0, 1, 2, 3\}\). Also remember that the Instantiate command that generates a prefab can be given both the position of the object as well as its orientation in the form of a quaternion.\(^1\)) Throughout the game, the walls swing 90° at random times. Here is how I suggest that you to implement this. At random times, but roughly once every 10 seconds, a wall will decide that it is time to start swinging.\(^2\) When it decides to swing, select a random direction (clockwise or counterclockwise), and initiate a process that causes the wall to execute a 90° rotation within 1 second. This can be done by applying a rotation of 90° \(\cdot\) Time.deltaTime during each update cycle. After the rotation is complete, the wall should again be aligned with either the \(x\)- or \(z\)-axis.

**Pick-ups:** Similar to the Roll-a-Ball tutorial, add some pick-ups placed randomly throughout the board. The pick-ups should be centered in the middle of the squares of the grid, that is, at positions \((i + 0.5, j + 0.5)\) for \(0 \leq i, j \leq 9\). Note that no pick-up should be placed on the goal position. As in the Roll-A-Ball tutorial, they should rotate (or have some other cool special effect of your own design).

The initial number of pick-ups should always be the same (we selected six) Beware of placing two pick-ups on the same grid square. Now, rather the proceeding straight to the goal, the player needs to hit some fixed number (we selected two) of the pick-ups. Please keep this number small, so that the TA can easily test your game.

**Text:** A new addition over Part I is that text should be displayed in your window. In the upper left, there should be a counter indicating how many pick-ups remain to be selected (e.g., “Pick-ups remaining: \(2\)”). Note that even if the user gathers an excessive number of pick-ups, the displayed quantity should never become negative. When the required number of pick-ups have been collected, the message should indicate this (e.g., “Done! Go to the goal!”). Finally, in the upper right, there should be a message indicating whether the game has been won (e.g., “Congratulations. You

\[^1\]\)Objects in Unity rotate about their center point, but each swinging wall rotates about the hinge point, which is off center. Here is a cute solution for tricking Unity into getting the rotation point over the hinge. Make your swinging wall a two-node hierarchy. The parent is an empty game object centered at the hinge, and the child is the wall, with its hinge aligned with the parent’s center point. When you apply a rotation to the parent, the wall swings about the hinge as desired.

\[^2\]\)This can be done by generating a random float using Random.Range \((0.0f, 1.0f)\) and checking whether the value is smaller than Time.deltaTime \(\sigma\), where \(\sigma = 10\) is the time between successive swing events. This test succeeds with probability \(1/10\) per second, and therefore the wall swings approximately once every 10 seconds.
win!”) or lost (e.g., “Sorry. You lose!”). As mentioned above, the only way to lose the game is to jump over the wall. We detected this when the $y$-coordinate became a sufficiently small negative number.

**Additional Notes:**

**Final Submission:** Final submission instructions will be posted on Piazza. (If you are ready to submit and do not see the instructions, please remind me.)

**Sample Executable:** I usually post a sample executable of my program on the [Class Projects Page](#).
I will make an announcement about this soon.

**Programming Style:** While we encourage clean programming structure, good structure is not an essential part of your grade. We reserve the right to deduct points for programs that are so poorly documented or organized that the grader cannot figure out how your program is working.

**Optional Elements for Extra Credit:** You may add additional features to your game for the purposes of extra-credit points. (See the syllabus regarding extra-credit points.) Please explain any additional features are in your `Readme.txt` file. The number of points of extra-credit credit will be left to the discretion of the grader.

**External Resources:** If you make use of any external resources in your program (or things that you developed prior to this class), even if you modified them, you must credit them in your `Readme.txt` file. Failing to do so will be considered an act of plagiarism. If you are unsure, check with me.