Programming Assignment 1: Simple Unity Game

Handed out: Tue, Feb 9. Due: Part I is due Wed, Feb 17, 11:59:59pm and Part II is due Wed, Feb 24 at the same time. 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 modifying the Roll-a-Ball tutorial on the Unity home page. There are two parts. The first is a very simple extension of the tutorial, and the second involves adding a new level with more complex behavior.

Part-I Requirements: Add the following modifications to the game given in the Unity Roll-a-Ball tutorial.

Variable pickups: In the online tutorial, there is a fixed number of pickups, and the game is won when you collect them all. Instead, there should be two public variables attached to some script in your program:

1. the initial number of pickups (say, numberOfPickups) and
2. the number of pickups to collect in order to win the game (say, pickupQuota).

This will allow the grader (from within the Unity editor) to alter their values and rerun your game. For example, you might add these public variables to your player-controller script. By setting the first to 15 and the second to 5, the game will start with a circle of 15 pickups, and you win the game as soon as you collect any 5 of them. (You may assume that pickupQuota ≤ numberOfPickups.)

Automatic pickup placement: In the tutorial, the pickups are placed by hand in the Unity editor. Instead, in one of your scripts you should automatically place the appropriate number (numberOfPickups) of pickups in a circle centered at the origin (where the player starts).

To do this, you can use the Instantiate function to create the pickups and Mathf.Cos and Mathf.Sin to determine the \((x, z)\) coordinates of these points. We used a circle of radius 7.5 in our implementation.

Count-Remaining Text: Rather than showing a count of the number of pickups that have been collected so far, instead print a count of the number of pickups that remain to be picked up before winning the game. This number should be displayed in the upper-left corner. As in the tutorial, when the game is won an appropriate message is shown in the middle of the window.

Quit/Rerstart: After winning, provide the user the opportunity to either restart the game from the beginning or to quit. (For example, they might hit ‘Q’ to quit and any other key to restart.)

Freezing on termination: When the game has ended (either by winning or losing) everything should freeze. That is, the player ball should not move and the pickups should stop rotating.
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).

**Notes for Part I:**

- 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 within the Unity simulator 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.

- Stopping the player from moving in physics mode is a bit tricky, since you need to eliminate all forces acting on the object. A simpler approach for causing a game object to stop moving is to remove it from the physics calculations by setting `rigidbody.isKinematic = true`, where `rigidbody` is the rigid-body component of the player. This puts the player under the full control of the program, and if you do nothing to change its position, it will not move.

**Part-II Requirements:**

**Multi-Level Structure:** The game should have two levels, the level from Part I and a second level. For the grader’s sake, there should be an easy way to bypass Level 1 and go directly to Level 2. It should also be possible to exit the game entirely at any time (say, by hitting the ESC key).

**Different Environment:** We modified the environment from Part I, creating a larger and more complex environment. First, the ground was reshaped to size $20 \times 30$, and we added gaps at the corners and at the centers of the two side walls. (There are effectively six walls.)

**Player Can Jump:** In addition to the existing horizontal motion along the ground, the player can also jump. We implemented this by adding a vertical force whenever the space key is hit. (This applies even if the player is already in the air.) If the player hits any of the other objects in the scene, it should bounce off of them. (see the [Unity Bouncing Ball tutorial](#) for information on how to do this using Physic Material.) If the player jumps outside of the ground area and falls so that the $y$-coordinate is negative, the game is immediately lost.

**Power-Up State:** The player can be in one of two states, *normal* and *powered-up*. We place a target above the center of the ground (a green disk in our implementation) that behaves like a trigger. If the player ball hits this trigger, it changes its appearance (we changed its color), and it transitions into the powered-up state for some fixed time period (we used 10 seconds). We used red for the normal state and green for the powered-up state.
The effect of pickup collisions depends on your state. When in the normal state, collisions with pickups are bad, and each collision increases the number of pickups that you need to collect by one. Further, the pickup object does not vanish. In the powered-up state, each collision with a pickup causes it to vanish.

**Pursuing/Evading Pickups:** The pickups move. When the player is in the normal state the pickups chase after the player object. This is done by generating a force vector in the direction of the player. When the player is in the powered-up state, the pickups run away from the player.

If a pickup falls off the board as a result of this, then it dies, and the player is given credit for collecting the pickup. (In our implementation, the pickup does not die right away. It dies once its $y$-coordinate is smaller than $-50$.)

**Jumping Beans:** In our first implementation we found the pickup movement to be very boring. It makes their behavior much more interesting to have them jump periodically. To achieve this, at random moments (at the rate of roughly 1 per second) a sufficiently large vertical force is generated to cause it to jump a small distance in the air. (Unlike the player, pickups must be on or at least near the ground when they jump.)

**Final Submission:** (Final submission instructions will be forthcoming.)

**Programming Style:** We will be reading your code to see that you implemented everything in a reasonable manner. Although programming style is not an explicit part of your grade, we reserve the right to deduct points for programs that are so poorly documented or organized that the TA 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 TA.

**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.