CMSC 425
Programming Assignment 1, Part 2
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.

Installment 1

Environment/Platform

The environment for Part 2 consists of the Platform, which is generated at run-time. Thus, the only thing that you will see in Scene view is the bowling ball.

The Platform is generated from a collection of tiles. Each tile is a Unity cube of dimension 10 x 1 x 10. Because we did not have use of the Unity physics engine, we did not need to associate a collider or rigid-body with each tile. We assigned the tile a dark blue color, RGB = (30, 50, 150).

Creating/Loading/Instantiating the Tile Prefabs

You will need to create prefabs that will be loaded at run time. Here is how to do this.

Create the prefab:

- First, use Create → 3D Object → Cube. Set its name to "Tile". Be sure to reset its position to (0, 0, 0). (See the Roll-A-Ball tutorial for how to reset newly created objects.) Scale it to the appropriate size and create/assign its material.
- To turn it into a prefab that can be loaded, first go into your Project window, and at the root level create a new directory called "Resources". Then drag your Tile game object into the Resources folder (or if you prefer, it can be in any subfolder within this folder).
- Delete the Tile from your Hierarchy window. It has now been saved as a prefab, and so you don't need the one in your Hierarchy any more.

Load the prefab:

- Before you instantiate your prefab, you need to load it. This is done with the following command:

```
GameObject tile = Resources.Load("Tile") as GameObject;
```

You only need to do this once. When the game object is loaded, you can instantiate as many copies as you like. This command will likely be placed in the script that builds your platform. Note that the Load command looks in the folder "Resources", so the argument "Tile" is the path leading to the Tile prefab within the Resources folder (that is, "Assets/Resources/Tile").

Instantiate the Prefab:

- We will be generating lots of prefabs. To keep them from cluttering up our hierarchy, let's make them the children of some empty game object. Let's create a Unity empty game object, and name it "Platform". We can attach our script for generating the platform to this game object.
- To build the platform, you will need to instantiate several tiles in the desired pattern. Given the layout suggested in the project handout, because the tiles are 10 x 10 in size, if you wanted to instantiate a tile at row (row) and column (col) of your platform grid, its center would be at the coordinates (10 * col, 0, 10 * row). That is, the z-coordinates correspond to the rows of the defining matrix, and the x-coordinates correspond to the columns of the defining matrix.
- To instantiate a prefab, we use Unity's Instantiate command. We need specify three things: the location of the instantiated object, its rotation (which for us will just be the identity), and its parent. Since our script for generating the platform will be attached to our empty Platform object, the desired parent will just be our own `transform`. Thus, we can use the following command for generating a tile at a given row (row) and column (col) using:

```
GameObject newTile = Instantiate(tile, new Vector3(10 * col, 0, 10 * row),
                     Quaternion.identity, transform);
```

Platform

As mentioned in the project handout, your platform should be generated by a script that inputs an array-based description of the platform and instantiates the appropriate set of tiles. In our implementation, we used an array of strings to do this:

```
string[] platformLayout = {
    "* ************",
    "* *   *       *",
    "* *   *   *   *",
    "* *   *   *   *",
    "* *********   *",
    "*     *   *   *",
    "*******   *****",
    "      *        ",
    "****  *        ",
    "*  *  *********",
    "*  *       *  *",
    "*  *****   *  *",
    "*  *   *   *  *",
    "*  *   *   *  *",
    "** *   *   *  *",
    " * *   *   *  *",
    " * *** *****  *",
    " *   *        *",
    " *   *  *******",
    " *   ****      " };
```

You should feel free to experiment with a platform of your own design. (Just don't make it so hard the grader can't win your game.)

Our platform has 20 rows and 15 columns. To generate the platform, write a couple of loops to run through the string array and instantiate tiles corresponding to the rows and columns having an asterisk. By the way, in C# you can access row r and column c in this string array using `platformLayout[r][c]`.

Because you will also need to refer to this array throughout the game, you may wish to generate your own representation of the structure for more convenient processing, but this is entirely up to you. For example, I created a parallel (20x15)-dimensional array where I stored a reference to the game objects returned by the Instantiate command. This might come in handy for other purposes. For example, if I wanted to change the color of the tile on which the ball is sitting, such an array of references would be useful.

Camera

For Installment-1, we just reused the camera controller from the Roll-A-Ball tutorial. It was placed 20 units above and 10 units behind the initial placement of the ball.

Note that this is not correct for Installment 2. Eventually, the camera will have to change its position so that it follows behind the ball object whenever it changes direction. If you want to get a head start on Installment-2, you may want to check out the Unity command, `Transform.LookAt`, which creates a rotation that will point the camera in any desired direction.

Lights

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

Ball

As in Part 1, the ball is a black sphere, scaled to 2 x 2 x 2. We set its initial position to (0, 1.5, 0), which is just above the origin so that it sits on top of a tile placed at the origin (row = 0, col = 0). To obtain a shiny appearance, we set the material's "Smoothness" setting to 0.75.

Even though we won't be making much use of physics, we will need to detect collisions with the bunnies in Installment-2. For this reason, we associated it with a RigidBody and gave it a sphere collider. We made this collider a trigger.

Controlling the Ball's Motion

In addition to creating the platform, the biggest challenge of Installment-1 is controlling the ball's movement. Since the ball is spherical, we do not need to worry about rotating the ball, just setting its position.

Since we do not rely on physics, we will need to create a script, call it BallController, to control the movement of the ball. This will require that we have access to the platform structure. Remember that Unity provides you with a mechanism for exchanging information between different game objects. For example, if you wanted your BallController to obtain access to the platformLayout string array, you

could create a getter function in your Platform object that takes care of this by defining a public function in your PlatformController script

```
public string[] GetLayout() { return platformLayout; }
```

and then from your BallController script, you could access this using:

```
GameObject platform = GameObject.Find("Platform");
PlatformController platformController = platform.GetComponent<PlatformController>();
string[] layout = platformController.GetLayout();
```

and now your ball controller has access to the platform layout array. (By the way, like Java, C# has a shorthand for generating getters and setters. Just Google it.)

State:

Controlling the ball's motion is mostly involved with maintaining information about the current state of the ball. This state will tell you everything that you need to know to determine where the ball moves from one update call to the next. The following items are relevant to the ball's current state:

- Current position, which can be extracted from `transform.position`
- Current tile row and column index within the platform
- Current direction of motion
- Position relative to the midpoint of the current tile (before or beyond)

There is some redundancy in the above (for example, if you know the current position, current tile, and direction of motion, you can determine whether the ball has yet to reach the tile's midpoint or is beyond the tile's midpoint). There are several possible ways of representing these entities, and at various times, one representation may be more convenient than another.

For example, we chose to represent the balls direction as an integer 0, 1, 2, 3 (for North, East, South, and West, respectively, where the z-axis points North and the x-axis points East). This was handy for representing turns, since a right turn caused the direction to change as direction = (direction + 1) % 4. But, it was less convenient for other tasks. For example, you might want to compute a unit vector, `directionVector`, that points in the direction that the ball is moving.

This assignment provides the opportunity to think about good object-oriented design, and you should give some thought to your design.

From Input to Movement:

Given the above state information, here one approach for determining how to update the ball's position. (Of course, you may organize your code differently, and that is fine provided that the same behavior results.) We found it clearest to assign the ball one of three different status types:

BEFORE: The ball has entered a tile, but has not yet reached the tile's midpoint

BEYOND: The ball has passed through the tile's midpoint

BLOCKED: On reaching the tile midpoint, the ball cannot proceed, and must wait until the player enters a valid turn request.

When the user provides a Left-turn or Right-turn input, the controller checks whether the turn is valid. A turn is valid if (1) the square into which the turn is requested exists within the platform, and (2) the ball's status is either BLOCKED or BEFORE. Invalid turn requests are punished by decreasing the ball's current speed. (We reduced it to 25% of its current value.) If a request is valid, we save this information for later processing.

Let oldPosition denote the previous position of the ball. If everything goes to plan, the ball will advance along its current direction vector by the product of its current speed times the elapsed time, that is:

```
newPosition = oldPosition + directionVector * speed * Time.deltaTime;
```

Of course, it will not make it there if the way is blocked. Here is how we suggest processing an Update event for the ball, based on the ball's status. (This relies on the assumption that the ball moves a relatively small distance with each update step.)

BEFORE: (Not yet reached the tile's midpoint)

- If newPosition is also BEFORE, then advance to newPosition
- Otherwise, we have just passed the tile midpoint.
    - If a valid turn has been requested, advance to the tile's midpoint, update the ball's direction based on the requested turn, and set the status to BEYOND.
    - If no turn has been requested but continuing forward is valid (since there is a tile ahead) then advance to newPosition, and set the status to BEYOND.
    - Otherwise (invalid to proceed), advance to the tile's midpoint and set the status to BLOCKED.

BEYOND: (Gone past the tile's midpoint)

- If newPosition is in the same tile as oldPosition, then advance to the newPosition.
- Otherwise, advance to newPosition but note the change in the current tile, and set the status to BEFORE.

BLOCKED: (Stuck at the tile's midpoint)

- If a turn is requested and it is valid (there is a tile in that direction), then update the ball's direction based on the requested turn, and set the status to BEYOND.

Object-Oriented Design:

This is a great opportunity to consider how to approach this problem from an object-oriented perspective. One or more of the following objects might be useful:

Platform – Encapsulates the structure of the platform, allowing you to query which row-column entries contain tiles are which do not.

Locator – Encapsulates a (row, column) pair of a single tile on the platform. It might be nice to also incorporate a direction as part of this. Such a class could have a function that answers queries such as "if the ball moves in direction X through this tile, does the adjacent tile exist in the platform?".

State – Encapsulates all the information relevant to the current state of the ball.

There is not one correct answer for how to design your program, but this is a matter to which you should give some thought. A good design should make it possible for you to express the ball's motion in terms of clear and natural operations applied to objects (like "direction.TurnRight()"), rather than obscure expressions (like "(direction + 1) % 4").

By the way, not all your classes need to be derived from Monobehaviour. If it makes sense for your class object to have an Update function (such as BallController) then clearly you need to derive from Monobehaviour. However, if you just wanted to create a class that encapsulates a tile's (row,column) location in the platform grid, then it need not be derived from Monobehaviour.

Speed:

Unlike the Roll-A-Ball tutorial, the ball's speed is not under the direct control of the player, but it is affected by the player's actions. By speed, we mean just the scalar value, not the direction.

We set a variable maxSpeed to 30 and an acceleration rate of 20. (I think that I said 10 in the handout, but I've changed it since.) If the ball is not blocked at its current speed is less than maxSpeed, we increase its speed by `acceleration * Time.deltaTime` (up to a maximum of maxSpeed).

Restarting, Quitting, Pausing, Slow-Motion

To restart a given level, you just need to load the scene. Since we have multiple scenes now, it is probably easiest to do this by name:

```
UnityEngine.SceneManagement.SceneManager.LoadScene("Part-1-Scene");
```

or

```
UnityEngine.SceneManagement.SceneManager.LoadScene("Part-2-Scene");
```

Recall that you can quit using:

```
Application.Quit();
```

Pause and Un-pause and Slow-Motion can be implemented by a quick-and-dirty (but less than perfect) method:

```
Time.timeScale = 1.0f; // normal time
Time.timeScale = 0.0f; // pause
Time.timeScale = 0.5f; // 1/2 slow motion
```

Before loading a new scene, reset the time scale back to 1.

Installment 2

Getting the Camera to Follow the Ball

The camera should follow the player by a fixed distance and at a fixed height above the tile surface. We placed our camera 20 units behind the player, and 30 units above, relative to the ball's current position. We then rotated the camera so that it is pointing directly to the ball's center. This only involves a few

lines of Unity C# code (which will go into your camera controller script), but I will go through the entire derivation, since it is a nice exercise in geometric programming.

To place the camera, we need to specify both its position and direction. As in the Roll-A-Ball tutorial, the camera's position is explicitly set in a camera-controller script based on the ball's position. Unlike Roll-A-Ball, the camera needs to follow the ball, and so we will need to take the ball's direction into account as well.

Let p_b denote the current position of the ball, let v_b be a unit-length vector that indicates the direction in which the ball is moving. In your solution to Installment 1, your ball controller will have access to both the ball's position and direction. You will need to provide the appropriate public getter functions so that your camera controller can access this information as well.

Our first task is to show how compute an offset vector v_c of the camera relative to the ball. The vertical (y) component of this vector will be fixed throughout (as 30 units above the center of ball). So, let us focus on how to compute its (x, z) components. First, let's consider the ideal state, where the camera is following directly behind the ball. (We will discuss smooth camera motion in the next section.) Suppose that we use our earlier convention where the ball's direction is given as 0, 1, 2, 3 for N, E, S, and W, respectively. Call this ballDirection. Then the clockwise angle (in degrees) of the ball's direct relative to due north is

$$\theta_b \leftarrow ballDirection \cdot 90.$$

(As the ballDirection varies from 0 to 4, this will vary from 0 to 360, as expected.) To compute the directional vector, we need to take the sine and cosine of this angle. For this we need to convert from degrees to radians. Unity provides a built-in scale factor for this, called `Mathf.Deg2Rad`. We can compute the (x, z) coordinates of the ball's horizontal direction vector as

$$v_b \leftarrow (\sin(\theta_b \cdot \texttt{Mathf.Deg2Rad}), \cos(\theta_b \cdot \texttt{Mathf.Deg2Rad})).$$

(Unity provides functions `Mathf.Sin` and `Mathf.Cos` to compute these functions.) Since we want the camera to trail the ball by 20 units, our desired camera offset vector is -20 v_b. If we take the y-coordinate into account as well, this yields an offset vector of

$$v_c \leftarrow (-20\sin(\theta_b \cdot \texttt{Mathf.Deg2Rad}), 30, -20\cos(\theta_b \cdot \texttt{Mathf.Deg2Rad})).$$

Given this vector, the position of the camera can be computed as $p_c \leftarrow p_b + v_c$. We use this to set the camera's `transform.position` in the camera's `LateUpdate` function, just as in Roll-A-Ball. Once the camera's position is known, we just need to determine how to rotate the camera so that it faces the ball. This latter computation can be done very easily, because Unity provides a function for this:

```
transform.LookAt(Transform target);
```

If you apply this to the camera's transform, it will rotate the camera to point at the given target point. Setting target to the ball's position yields the desired camera rotation.

Smooth Camera Motion

There is one major problem with the above solution. Since the ball can change direction instantaneously, this results in a jarring jump in the camera's position whenever the ball changes

direction. Instead, we want the camera to rotate gradually into the proper position after each direction change. It is very easy to modify the above code to achieve this. The key is that when ballDirection changes, we do not want to immediately change the value of θ_b. Instead, we will gradually interpolate from the current rotation angle to the desired one.

To do this, let us save the current camera rotation angle in a variable θ_c. Then, the next time that `LateUpdate` is called, we compute the value of θ_b as before, but rather than use it to rotate the camera directly, instead we interpolate between the saved value, θ_c, and the desired new value, θ_b. Normally, we would use `Mathf.Lerp` to do this, but because we are interpolating angles (in the range from 0 to 360 degrees) and we want to avoid issues of the discontinuities close to 0 and 360, we use `Math.LerpAngle`. The amount of interpolation is a number between 0 and 1 and is given in the third argument. It should be sensitive to the value of `Time.deltaTime`, but you can include a scalar multiplier to adjust how fast the angle turns.

$$\theta_c \leftarrow \texttt{Mathf.LerpAngle}(\theta_c, \theta_b, \texttt{angleSpeed} * \texttt{Time.deltaTime});$$

We set angleSpeed to 2 in our implementation. Given this angle, we modify our computation of the offset vector to:

$$v_c \leftarrow (-20\sin(\theta_c \cdot \texttt{Mathf.Deg2Rad}), 30, -20\cos(\theta_c \cdot \texttt{Mathf.Deg2Rad})).$$

Otherwise, everything from the previous section is the same.

Chasers

Our chasers consisted of a group of 10 copies of the bowling pin object from Part 1. We placed them under a common empty game object in a rather arbitrary set of positions (since we didn't want them to look like a triangle of bowling pins). We centered the group one tile behind the ball, with the (x, z) coordinates of (0, -1). The y-coordinate was adjusted so that the pins sit on top of the tiles.

As with the ball, the chasers start with a speed of 0, and they accelerate in the same manner as the ball (see the section "Speed:" above) but their maximum speed was only 28 (in contrast to the ball's maximum speed of 30).

We offered two possible solutions for the chasing behavior:

Impulsive Chasing – The chasing game object simply moves toward the current ball location. While this is not very realistic, it is very easy to code. Here you can employ the following handy Unity function. Let `currentPos` be a `Vector3` that gives the current position of the chaser object, let `target` be a `Vector3` giving the desired target location (i.e., the ball), and let `speed` denote the desired speed. Then:

```
Vector3.MoveTowards(currentPos, ballPos, speed * Time.deltaTime);
```

Returns a `Vector3` containing the result of moving from the current position towards the ball's position at the given speed.

This method is suitable for partial credit, but it is not realistic, since the chasers can walk right off the platform. (We disabled gravity, so they don't fall.)

Path Following – The better way to solve the problem is to have the chasers following the same path as the ball. The approach is described in the assignment handout, which we quote below:

This can be done as follows. A queue data structure is created. Each time the runner passes through the center of a tile, we enqueue information describing the runner's action (turn-left, turn-right, or no-turn). Then when the chaser reaches its next tile (which the runner may have passed many tiles earlier), we dequeue the turn information and execute the desired turn. Thus, if the player is three tiles ahead of the chaser, there will be three entries in the queue. (Note that C# provides a generic Queue data type, so no need to implement your own.)

Since the chaser does not start on the same tile as the runner, this queue should be initialized with the turns needed to get to the runner's initial tile. (It is not necessary to implement a general solution. In our implementation, we assumed that the chaser always starts one tile behind the runner, and we initialized the queue with a single entry ``no turn''.)

Rotating the Chasers to Face the Ball

We also want the chasers to rotate so that they face the ball that they are chasing. This can either be done to the parent chaser object (so the chasers rotate as a group) or individually to each chaser (so each rotates in place). We applied it to the parent object.

The idea is to compute a vector from the chaser object towards the ball object, call it the chasing direction vector, and then apply a rotation about the vertical (y) axis to align the chaser's forward vector with this chasing direction vector. One easy solution is to apply the `LookAt` function, which we introduced earlier for the camera:

```
transform.LookAt(Transform target);
```

where the target is the ball's transform.

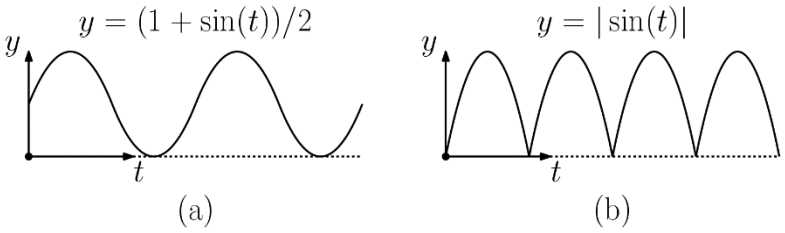
The result is not bad, but when the pins get very close to the target, they can lose their vertical orientation and slant upward or downwards, depending on whether the ball's center is higher or lower than the pin's center. An alternative approach, which you may prefer, is to use the Unity function that generates a rotation that gradually turns towards a given location. This function is called `Vector3.RotateTowards`. We used this approach in our implementation. (It is combined with the function `Quaternion.LookRotation`. You can check these out in the Unity documentation.) We explicitly set the y-component of the directional vector to 0 to avoid the vertical slant issue, which the LookAt method suffered from.

Getting the Bunnies to Hop

Although it is not a requirement of the assignment, it is nice to add some additional movement to the bunnies. We chose to add a simple hopping animation. The idea is very simple. Associate a script with each individual bunny that modifies the y-coordinate of the transform's position as a function of time. This can be done by adding the following lines to your Update function for each pin:

```
Vector3 newPos = transform.position;
newPos.y = JumpHeight(Time.time);
transform.position = newPos;
```

All that remains is to define a appropriate function JumpHeight, which changes the height as a function of time. JumpHeight should be a periodical function of time. You might consider using a sinusoidal function, but this will look a bit more like the smooth up-and-down movement of a horse in a carousel. To get something that looks more like hopping, you want a function that changes instantaneously when it hits the ground. The graphs below illustrate two examples of such functions, where the function shown in (b) is more like a hopping behavior. (We used a quadratic function in our implementation, but the sinusoidal function is easier to implement.)



To avoid having all your bunnies jumping in synchronization, you can add some additional randomness. For example, the function $\alpha \sin(\beta t + \gamma)$ has an amplitude (height) proportional to α, a frequency proportional to β, and a horizontal offset proportional to γ. By making these three parameters random properties of each bunny, their motions will differ from each other.

(More to come, stay tuned.)