CMSC 425 : Fall 2018 Dave Mount

Programming Assignment 1: Bowling-for and Running-from Bunnies

Due: Part 1 is due Thursday, Sep 13, 11:59 pm (25 points). Part 2 has two due dates. Installment 1 is due Thursday, Sep 20, 11:59 (10 points), and the final installment is due Tuesday, Oct 2, 11:59pm (65 points).

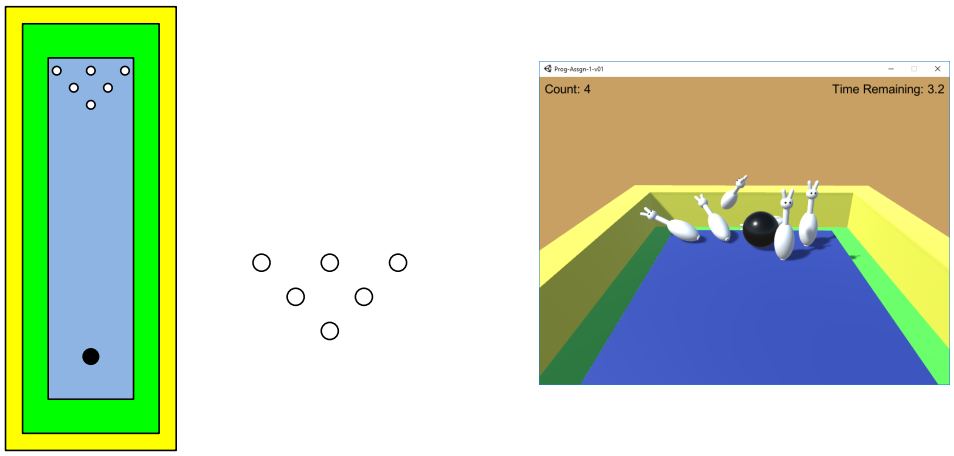
Late policy: Up to 6 hours late: 5% of the total; up to 24 hours late: 10%, and then 20% for each additional 24 hours. Submission instructions will be given later.

The purpose of this assignment is to learn the basics of Unity by making a simple game. The first part will involve a modification of the Unity Roll-A-Ball tutorial. The second part involves turning this into a 2-level game, where the second level is a runner-style game.

Don't Panic: Yes, this description is long. But the code is not as long as you might think. (There are just lots of little details.) Since there are many parts, you can get partial credit even if all the features are not implemented.

Part 1: (Bowling for Bunnies) Make the following modifications to the Roll-A-Ball tutorial, which will look more like a bowling game.

Bowling Lane: The game takes place on a long flat rectangular surface that resembles a bowling lane (shown in blue in Fig. 1(a)). (Rather than using a Unity plane, we recommend that you use a Unity cube that will be appropriately scaled.) Around all sides of the lane is a trough into which the ball will be trapped if it falls off the lane (shown in green in Fig. 1(a)). This trough is surrounded by wall, which are designed to keep the ball from leaving the gutter, once it lands there (shown in yellow in Fig. 1(a)).



(a) (b) (c)

Figure 1: (a) Top-down view, (b) Pin setup, (c) Screen capture of game in action.

Bowling Pins: At one end of the lane there is a set of 6 bowling pins arranged in the usual triangular pattern (see Fig. 1(b)). (We have decorated ours up to look like they have cute bunny heads. You can be creative in how yours look, but each pin should consist of an amalgamation of three or more Unity primitive shapes.) We want the pins to behave in a lively manner when they are hit by the ball. You can do this by associating all your colliders with a Unity Physic Material with a fairly high "bounciness" value. We used 0.5.

Ball movement: The ball moves using the same controls as in the Roll-A-Ball tutorial.

Winning: Pins can be knocked over either by the ball or by other pins. When a has first tilted beyond 45° relative to the vertical direction, it is considered to have "toppled". The game is won when all six pins are toppled. As in Roll-A-Ball, there should be a message in the upper left that indicates the number of pins that have been toppled. (Note that a pin may be hit, but may not topple until significantly later.) When all six pins have toppled, a congratulatory message is shown in the center of the window.

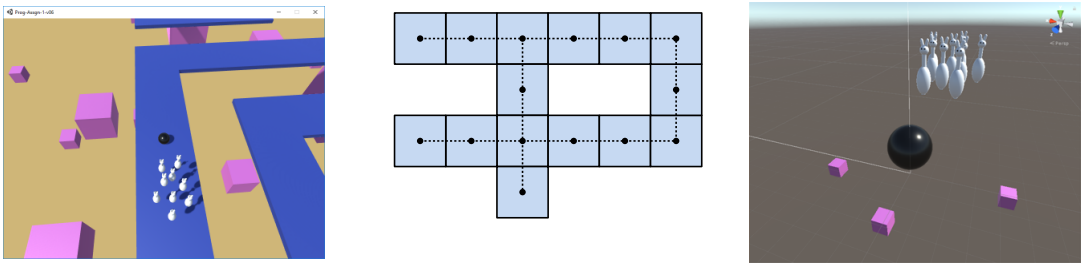
Camera: As in the Roll-A-Ball tutorial, the camera should follow the ball, but only to a limit. (Otherwise it is hard to see the pins.) Once the ball passes an imaginary curtain through the frontmost pin, the camera should stop following.

Timing Out: If the pins do not all topple after 10 second have elapsed, a message should be displayed in the center of the window indicating that the game has been lost. If the last bunny topples after 10 seconds, the game is still considered lost, and this message should not change. A countdown timer should be shown in the upper right, which displays time to a resolution of tenths of a second.

Restart/Quit: If the user hits the 'R' key, the game restarts immediately. If the user hits the 'Q' key, the game quits. (Note that a game will not quit when run within the Unity engine or as a WebGL program. This can only be tested by building an executable.)

Part 2: (Bunny Runner) In Part 1, we let Unity's physics engine do most of the work for us. In this part, we will implement a style of game where the programmer controls the game's behavior. Consequently, there will be more programming in this part. We will implement a runner-style game (such as Temple Run) where the bowling ball, which is controlled by the player, is being chased by a horde of crazed bowling pins through a maze-like structure.

Overview: The game's principal elements are the bowling ball, called the *runner*, a horde of pins, called the *chaser*, within an maze-like set of paths floating in the air, called the *platform* (see Fig. 2(a)). The player controls the turns (left, right, straight) and the game controls the speed. The chasers are clustered into a group and chase after the runner. The runner's maximum speed is slightly higher than that of the chasers, but if he makes any errors (e.g., failing to execute a valid turn at the proper time) the speed is reduced significantly. If the chasers catch up with the runner, the game is lost. If the runner makes it to a special *target location* before this happens, the game is won. To add a sense of visual depth, we have added a number of purple cubes beneath the platform, but these play no roll in the game.



(a) (b) (c)

Figure 2: (a) Screen-capture of the game, (b) graph structure defined by the tiles, and (c) sample of scene in the Unity editor.

Platform: The platform consists of a maze-like structure. We implemented this as a grid of square *tiles*, where each tile is a $10 \times 1 \times 10$ Unity cube. The runner can move only along a path that

runs down the center of the tiles. This implicitly defines a graph structure, where the centers of the tiles are the nodes of the graph, and there is an edge between two nodes if their tiles are adjacent (see Fig. 2(b)).

The tiles are prefab objects and are generated at run time. (For example, in Fig. 2(c) you do not see the tiles in the editor until the game starts running.) In our implementation, the tile structure was specified by an array of strings whose entries define where there are tiles (indicated by $*$) and where there are none (indicated by a space). The string-array representation and actual layout are shown in Fig. 3. (Note that the z-coordinate corresponds to the array's row index and so the layout appears to be upside-down relative to the array form.) Your platform should be similarly configurable, by allowing the programmer to alter the platform structure within the program itself, rather than placing tiles in the Unity editor.

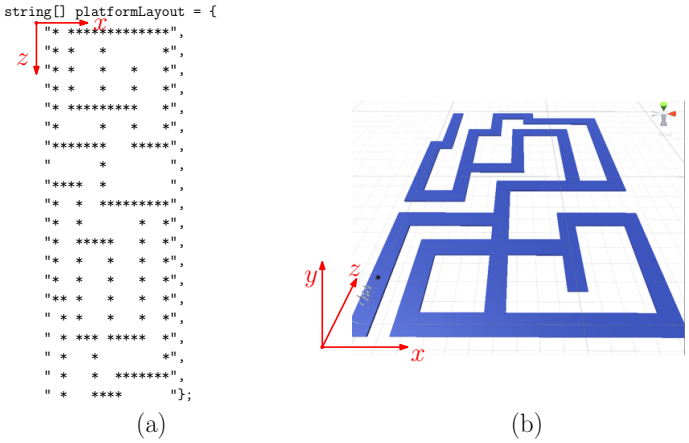


Figure 3: (a) Platform as a string array and (b) geometric layout.

Runner: The runner's speed is controlled by the game. The player determines when the turns are to be made. A left turn is requested by pressing the 'A' or '←' keys, and a right turn by pressing 'D' or '→'. The request must be made between the time that the runner enters the tile until it reaches the center of the tile. If the request arrives too late or if the requested turn is invalid (since there is no neighboring tile to that side) the runner's speed is decreased by some fraction (we used 0.25). If the runner arrives at a T-junction and fails to make a turn selection, the runner comes to a stop at the center of the square until a turn is entered.

If a turn is requested, then the turn is performed when the runner arrives at the center of the tile. (More accurately, at the first Update call when the runner crosses the center of the tile.) The runner is then placed at the center of the tile, and the direction is modified accordingly.

The runner starts with a speed of zero. The runner's speed does not change instantaneously. It increases at some rate (we used 10 unity units per second) until it reaches the maximum allowed speed (we used 30 units per second). Thus, from a stationary position, it takes the runner 3 seconds to achieve full speed. When a turn is made, the current speed is maintained.

By the way, we did not implement a command to perform a "U-turn", but this option could be added. Since this will lead the runner back at the chasing horde, it would be suicidal unless you invent an evasion technique, like jumping over the horde.

For Installment 1, you only need to implement the platform and the runner behavior.

Camera: The camera is always directed towards the runner and ideally it follows the runner from a fixed offset. (We placed our camera 20 units behind and 30 units above the runner.) The

runner turns instantaneously, and this would result in a jarring effect. So, we have the camera's actual position rotate smoothly around the player. This can be done by interpolating between the camera's current rotation and runner's orientation angle. (Since this is a rotation about the vertical axis, only the y-Euler angle is involved. Unity provides a useful function LerpAngle, which deals with the messy issue of correctly interpolating angles close to the discontinuity between $0°$ and $360°$.)

Environment: We used just a single directional light source as Part 1. Also, as in Part 1, we replaced the default Unity sky box with a solid color. To create a sense of depth, we added (by hand) a number of light-purple cubes that float beneath the platform. You are not required to make this changes, but you should replace the default Unity skybox and you should create some additional environment to give a sense of depth to scene.

Chasers: The bowling pins from Part 1 form a horde that chases the runner around the maze. There are a number of ways of implementing this. (Later in the semester, we will discuss *flocking behavior*.) We will adopt a simple approach. The individual chasers will be placed within an empty game object, called the *chaser*, and the chaser's motion will be controlled by an associated script.

The chaser starts at a speed of zero, and it accelerates at the same rate as the runner. The chaser's maximum speed is slightly smaller than the runner's maximum speed, so the player can build up a safety cushion if the player makes no mistakes. (We used a speed of 30 for the runner and 28 for the chaser.)

We will describe two methods for controlling the chaser's turns.There two options for how you handle the chaser motion:

Impulsive chasing: In this version the chaser just moves directly towards the runner, with no concern for where platform is located. With each update cycle a vector is computed between the center of the chaser object and the runner. With each update cycle, the chaser moves along this vector by its current speed. This is very easy to implement (and it is probably a good idea when you are debugging your game), but it is not very realistic. When the player makes a turn, the chaser can take a "short cut" by flying off the platform. (I did not have the chasers affected by gravity, so they just appear to fly across the gap.)

Path following: In this version the chaser replicates all of the turns that the runner makes. 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".)

There will be a small point deduction if you implement the impulsive-chasing rather than path-following, but as mentioned earlier, it would be a good idea to start with impulsive-chasing (which takes about three lines of code) and then upgrade to path-following (which takes about a page of code).

In addition to following the runner, the chaser objects should reorient themselves so that they always face the player. This can be done in two different ways, your choice. Either the entire chaser object rotates as a unit or the individual chaser objects maintain the relative positions and rotate individually.

Winning/Losing: If the chaser catches up with the runner, then the runner loses. On the other hand, if the runner reaches a designated *target tile* before this happens, the runner wins. In either case you should display an appropriate message and freeze the game graphics.

Transitions: In addition to the requirements from Part 1 to restart and quite the game, you should provide the capabilities to jump between the levels, pause the game, and (for Part 2) to run the game in slow-motion. While you are free to do this however you like, here is a proposal for the sake of uniformity:

'Q'	quit	'R'	restart current level
'P'	pause/unpause	'Z'	slow-motion/regular-speed
'1'	load Part 1	'2'	load Part 2

Your slow-motion does not to be perfect. A simple solution is to set Time.timeScale to 0.5.

Final Submission: Final submission will be by uploading a file through ELMS. (We do not use the submit server.) Detailed submission instructions will be posted through Piazza. If you are ready to submit and do not see the instructions, please remind me.

Sample Executable: We have posted a sample executable in the Projects page of our class web page:

http://www.cs.umd.edu/class/fall2018/cmsc425/projects.shtml

This is just for guidance. You are not required to mimic our exact look and/or behavior.

Common Questions:

- **"Does my implementation have to look/behave exactly like yours?"**
  No. In fact, you are encouraged to make creative changes to suit your own taste, provided that your submission satisfies the spirit of our requirements and achieves the same learning objectives. If you are wondering whether your modifications are acceptable, please check with your instructor at least 24 hours before the due date.
- **"Will you deduct points for poor programming style?"**
  Possibly. While we encourage clean programming structure, this will not constitute a major part of the grade. Most of you will be new to Unity programming, and we will be forgiving of awkward structure, especially in the first assignment. Nonetheless, 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 works.
- **"Can I get extra credit if I exceed your requirements?"**
  Yes. We ask the graders to assign credit for additional work that goes beyond our basic requirements. These *extra-credit points* are not part of the assignment score, but instead are recorded separately. At the end of the semester, final grade cutoffs are determined without consideration of these extra-credit points. Thus, your grade cannot be negatively influenced by not doing extra-credit work. However, if your final score is just below a cut-off between two letter grades, we may take these extra-credit points into consideration before assigning the final course grade.
- **"Can I make use of resources that I got from elsewhere?"**
  Yes, provided these resources to not circumvent the assignment's learning objectives and provided the you cite where you got the resources. For example, if you downloaded a cool model from the Unity asset store or if you implemented something you learned from an online tutorial, you *must* tell us your sources. (To otherwise would be taking credit for someone else's work. Doing so will *not* affect your grade negatively, but failing to do so may result in disciplinary action.) You are *encouraged* to tell us how you modified it to make it work, since we would like to give you credit for your effort.