CMSC 425 : Fall 2018 Dave Mount

Programming Assignment 2: Evil Elvis vs. Granny

Due: Wednesday, Nov 28, 11:59pm (100 points total - The point values indicated below are approximate and subject to change).

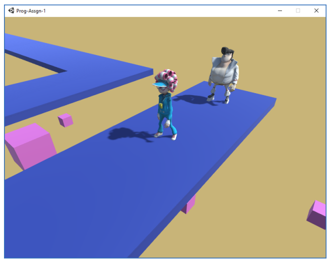
Late policy: Up to 6 hours late: 5% of the total; up to 24 hours late: 10%, and then 20% for each additional 24 hours. Submission instructions will be given later.

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.

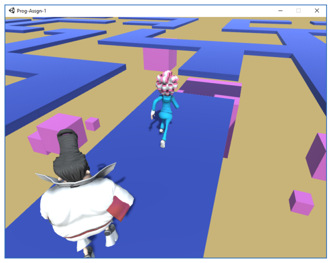
The purpose of this assignment is to enhance our earlier programming assignment and produce a much more complete game. There are three major new elements to this assignment:

- Animated characters - The rolling ball and chasers will be replaced with animated characters that will run around the platform.
- Jumping - There will be gaps along the straight portions of the platform that the running character must jump over.
- **Optional:** Automatic platform generation - When the game is restarted, the platform will be generated randomly.

Overview: The game's principal components are the same as in the earlier programming assignment. The player controls a running character (Sporty Granny, in our implementation), called *runner*, that is being pursued by an autonomous character (Elvis, in our implementation), called the *chaser*. They run along a maze-like set of paths formed by square tiles that are floating in the air, called the *platform*. (See, Fig. 1.) The platform has gaps, which the runner must jump over (or else fall to her death).



(a)



(b)

Figure 1: (a) Starting view of Sporty Granny and Elvis and (b) screenshot from the middle of the game.

The player controls the turns (left, right, straight), by hitting '←' or '`A`' to turn left and '→' or '`D`' to turn right. The runner's speed is controlled by the game. The runner's maximum speed is slightly higher than that of the chaser, but if she makes any errors (e.g., failing to execute a valid turn at the proper time) the speed is reduced significantly. If the chaser catches up with the runner, the game is lost. If the runner makes it to a special *target tile* before this happens, the game is won.

Animated Characters: (25 points) Both the runner and the chaser are to implemented using animated models. (We will provide you with both sample characters and animations, which we obtained from Mixamo, but you are allowed to find your own. As usual, you must cite all your sources.)

In order to receive full credit, your game must implement at least the following different animations or their rough equivalents: *idle*, *walking*, *running*, and *jumping*. For extra credit, you may optionally include other animations, such as one to show when the runner falls off the platform, and one to show when the game is won or lost.

Blended Animations: (25 points) In order to produce realistic looking motion, you should use Unity's tool, called *Mecanim*, for blending between animations. The transitions should be smooth enough and slow enough that the grader can verify that you implemented this properly.

As a guide (but not a requirement) we set the runner's maximum horizontal speed to 20 units/sec and the acceleration rate to 10 units/sec^2. We used the following transitions in our animator, based on the horizontal component of the speed:

- 0.0 – 0.1 units/sec: Idle
- 0.1 – 10 units/sec: Walking
- $>$ 10 unites/sec: Running

This means that it takes around one second to transition from idle up to running, which is sufficient. Transitions should occur promptly when they are requested. (If your transitions are happening after some delay, check that the "`Has Exit Time`" check box has been unchecked in your animation transitions. Only the jumping animation should have this box checked.)

In order to make is possible for the grader to clearly see these transitions, it will be necessary to have a mode in which the chaser is disabled, allowing us to follow the runner around the platform. When the user hits the '`C`' key, the chaser should not move. (You may implement however you like. You can simply have him stop moving, or you can destroy or disable the chaser game object completely. In our implementation, the '`C`' key acts like a toggle, causing the chaser to alternate between being stopped and moving.)

Platform Gaps: (5 points) To add a bit more challenge, insert occasional 1-tile gaps along some of the straight portions of your platform. The runner and chaser can move through these gaps, but if the runner ever walks, runs, or lands within a gap tile, it falls through and loses the game. In contrast the chaser can simply run in mid air through these gaps. (It is not required to create a jumping animation for the chaser, since this movement is rarely visible to the player.) For the sake of movement dynamics, gap tiles behave the same as regular tiles, but no game object is generated. If the runner ever steps onto a gap tile, the game is lost.

In order to specify where these gap tiles are, we found it convenient to modify our platform generator code. Recall that it takes a 2-dimensional string array as input. We replaced some of the '`*`' characters with '`-`' to indicate a gap tile in the layout. We also used '`S`' to specify the start tile for the runner, and '`T`' to indicate the target tile. (This is not a requirement.) Remember that the player moves initially to the north, and the chaser starts on tile behind the runner. An example of the one that we used in our game is shown in the code block below.

Jumping: (25 points) Implement a jumping command (using the space bar) to allow the runner to jump over gaps. When the space bar is hit, the runner should simultaneously jump up into the air and should also blend into a jumping animation. (For partial credit, you can implement the jumping motion, but without the animation.)

Beware that many jumping animations (include the Mixamo jumping animation that we provide) creates the illusion that the player is jumping, but the collider stays in contact with the ground. You will need to actually change the runner's vertical position in your script to simulate real flight.

We do not allow jumping to occur at all times. First, the runner has to be running (horizontal speed of at least 10 units/sec), and she must currently be in contact with the ground. Thus, you cannot jump while walking or idle, and you cannot jump if you are already in the air.

Sample Platform Layout

```
string[] platformLayout =
{
    "* ********-****",      // * = regular tile
    "S *   *       *",      // - = gap tile
    "* *   *   *   *",      // space = no tile
    "* *   *   *   -",      // T = target tile
    "* *********   *",      // S = runner start tile (initial direction is north)
    "*     *   *   *",      // Chaser starts one tile to the south of the runner
    "***-***   *****",      //   |
    "      *        ",      //   |
    "****  *        ",      //   V  This way is north
    "*  *  ***-*****",
    "*  *       *  *",
    "-  *****   *  *",
    "*  *   *   -  *",
    "*  *   -   *  *",
    "** *   *   *  -",
    " * *   *   *  *",
    " * *** *****  *",
    " *   *        *",
    " *   *  ***-**T",
    " *   ****      "
};
```

You might think that you need to use Unity's physics engine to do the jumping for you, but this is not a good idea. You will want to be able to control the exact height and time in the air. It is not hard to implement your own gravity, without using rigid bodies or colliders. See the "Tips on Jumping" at the end of this handout.

Falling: (5 points) The game is lost if the runner is on the ground over a gap tile (either by moving into the tile or landing from a jump within the gap). Rather than just freezing the game at this point, let it go on at least a few seconds longer to make it clear that the runner has fallen off the platform. (Because we do not implement physics, it is possible for the runner to fall through solid objects along the way. This is okay.)

General Requirements: (15 points) A number of elements from the earlier programming assignment should be included as well. These include:

- Camera: The camera should following the player, making smooth rotational transitions
- Background: There should be some background to create a sense of depth for the platform. Also, the default Unity skybox should be replaced (e.g., with a nicer skybox or a solid color).
- Winning/losing: There should be some indication of winning/losing, for example, by freezing the game and displaying an appropriate text message
- Transitions: You should implement transitional commands, including '`R`' for Restart, '`Q`' for Quit, '`Z`' for Slow motion, '`P`' for Pause/Unpause, and '`C`' to disable the chaser. You do not need to include a start menu, and you do not need to include an implementation of the first programming assignment.
- Submission format: Follow the standard submission instructions from the first assignment. Remember to remove everything except your `Assets` and `ProjectSettings` folders. Also, remember to include your `ReadMe.txt` file including instructions, known issues, additional features for extra credit, resources used.

Random Platform Generation: (Optional - For extra-credit points) Every time the game is restarted, generate a random platform. Here is a suggestion for how to do this.

First, fix the dimensions of your desired platform, making them a multiple of some small integer k, say from 2–5. (We'll see why below.) Set $R = kr$ be the number of rows and $C = kc$ be the number of columns, for some integers r and c. Next, generate a random maze on an $r \times c$ grid. We will discuss ways of doing this in class, but one of the easiest ways it to apply a depth-first search algorithm to produce a DFS tree for the associated $r \times c$ grid graph. (This tree is shown with dashed edges in Fig. 2(a). We chose $r = c = 6$.)

There are two problems with using the resulting graph directly. The first is that it will naturally have a lot of dead-ends in the maze. A dead-end is a graph vertex of degree one, or equivalently a square of the grid bounded by three walls (shown in blue in Fig. 2(b)). These are problematic, since if the player enters a dead-end, it will surely lose the game. Fixing this is pretty easy. For each dead-end in your maze, remove any one of these walls (or equivalently join it with any of its neighbors). (See Fig. 2(c) for the result.)

The second problem is that your maze is way too dense to make for a good platform. The solution is to expand the grid to its full $R \times C$ size, and replace edge of the original maze with a sequence of k tiles (see Fig. 2(d)). If this is not dense enough, then just use a larger scale factor for k. (By the way, the size is not exactly $R \times C$; it is really $(R - k + 1) \times (C - k + 1)$.)

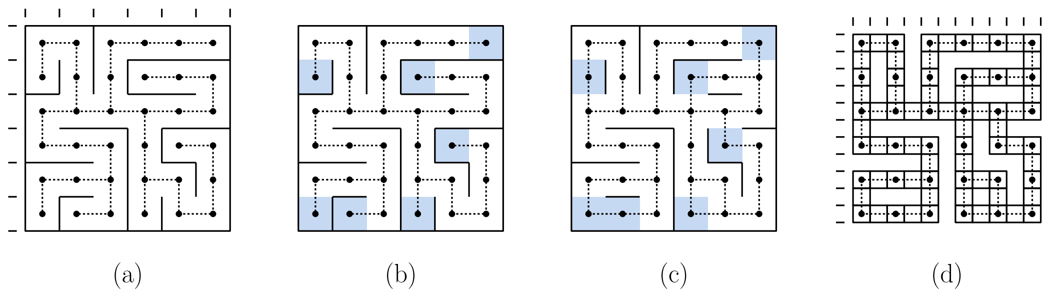


Figure 2: Building a random platform for $r = c = 6$ and $k = 2$.

You need to select the starting square and the target square. If you engineer the DFS so that it starts with the pair $[0, 0]$ and $[1, 0]$, you can use these. The target can be chosen at random. Also, you will want to insert random gaps. This can be done by selecting a few edges at random and removing a random tile in the middle of each.

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

Sample Source and 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.

We will make available a limited implementation of our Programming Assignment 1. You are allowed to use that as a starting point if you like.

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. Now that you have had a bit more experience with Unity, we will be expecting that your solutions will be cleaner than in the first assignment. While style will not be a major part of the 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 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. For example, downloading a 3rd-person controller is not allowed, because this is an important feature of the assignment. But you can certainly find your own models and animations.

  Remember that must **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.

Tips on Jumping: Because the game does not employ physics (assuming you implemented yours as we did), I would suggest not using the Unity physics engine to do this. It is easier to implement on your own and gives you better control of the runner's state. Here is the idea. For the earlier assignment, you maintained the runner's horizontal position and speed. For this part, you just need to maintain the runner's vertical offset and speed.

First, define the *platformTop* to be the *y*-coordinate of the top of the platform. (In our implementation, this was half the thickness of the platform tiles.) Define the *vertical offset* to be distance of the runner's feet above this level. Define the *vertical speed* to be the vertical component of its speed. When the vertical speed is positive, the runner is ascending and when it is negative she is descending. Initially, both the offset and speed are set to 0.

When the space bar is hit, set the vertical speed to an appropriately large positive value (we set it to 9.0). We also set a boolean status variable, `inFlight` to let us know that the runner is in the air. Then, each time the Update function is called, we do two things. First, we decrease the vertical velocity according to the effects of gravity, and second we change the vertical offset according the speed times the elapsed time. In summary:

```
float platformTop = ... // set this to the y-coordinate of the top of your platform
float gravity = 20.0f; // acceleration due to gravity - adjust as you like
float jumpImpulse = 9.0f; // vertical speed increase when jump requested - adjust as you like

void Start() {
    // ...
    verticalOffset = verticalSpeed = 0.0f; // initially on the ground
    inFlight = false;
}

void WhenSpaceBarIsHit() {
    verticalSpeed = jumpImpulse; // add a sudden impulse to vertical speed
    inFlight = true; // we are now airborne
}

void Update() {
    // ...
    if (inFlight) { // in the air?
        verticalSpeed -= gravity * Time.deltaTime; // gravity decreases vertical speed
        verticalOffset += verticalSpeed * Time.deltaTime; // adjust offset based on vert speed
    }
    transform.position = currentHorizontalPosition + Vector3.up * verticalOffset + platformTop;
    // ...
    if (transform.position.y <= platformTop) { // back on the ground?
        verticalOffset = verticalSpeed = 0.0f;
        inFlight = false;
    }
}
```