Assignment 1 - Building a Three Dimensional Maze
Due. Friday, January 29, 2021

We are going to recreate the Rogue game in three dimensions. Each assignment in the course will add more features to the game. This assignment focusses on creating the maze and some objects in the world, as well as adding collision detection and gravity for the player.

A description of Rogue is available here:
https://en.wikipedia.org/wiki/Rogue_(video_game)

You can play Rogue in a web browser at:
https://www.myabandonware.com/game/rogue-4n/play-4n

A description of the Rogue-like games is available here:
https://en.wikipedia.org/wiki/Roguelike

1. World Building

Create a three dimensional maze from cubes.

The mazes in Rogue consist of several rooms attached by corridors. The algorithm to create the maze involves randomly creating the rooms and then attaching them to each other using the corridors. The rooms are organized in a 3x3 grid.

1. Create nine randomly positioned rooms. The rooms should have randomized depths and widths.
2. Randomly position doors on the room walls. You will need one door for each corridor to another room. It makes sense to place the doors on the walls facing other rooms. Do not align the doors so there is always a straight path between them.
3. Attach the rooms to each other by building corridors between the doors.

The maze must be randomly generated. Each time the program is run it should produce a different maze. The maze should consist of nine rooms with three rooms per row, with three rows.

The rooms can all be rectangular or square. You can experiment with other shapes for rooms is you wish.

Corridors are created using the doorways between two rooms as the starting and ending positions. If the doors are not directly facing each other then you will need to create a bend in the corridor so it can attach between the two doors. The easiest way to do this is likely to pick a space between the two doorways and create a perpendicular corridor to join the two doorways.

For example, if these are the edges of two rooms and the d is the doorway.

```
             XXX
XXX          X
  d          X
  X          X
  X          X
XXX          X
             d
             XXX
```

Each corridor will move away from the door in a line.

```
             XXX
XXX          X
  d......    X
  X          X
  X          X
XXX          X
      .......d
             XXX
```

Pick a random perpendicular corridor to join these two corridors.

```
             XXX
XXX          X
  d......    X
  X     .    X
  X     .    X
XXX     .    X
        .....d
             XXX
```

You should calculate the locations for the perpendicular corridor before any corridors are created.

There must be enough space between each room to allow a corridor to pass between them. This means that rooms cannot be created with only one row of blocks between them.

Make the rooms and the corridors at least two units tall. This will be necessary to test gravity.

Make the walls of the maze a different colour to the floor so they are clearly visible. You can create custom colours for the cubes using setUserColours() if you wish. Don't choose colours that make the maze difficult for the TA to see (such as making the walls, floor, and ceiling all the same colour).

Create a pattern in the ground using two different colours. For example, the floor could be primarily dark brown with some lighter brown squares which contrast the other areas. This can be done by changing the colour at different randomized locations or by creating a pattern on the floor.

Position the maze near the middle of the world array (somewhere around 25 in the y axis of world[x][y][z]). We may add multi-layer mazes with higher or lower level in the future so there will need to be space in the world array above and below the current level.

Place some randomly positioned cubes on the ground in the rooms. These should be single cubes. There do not need to be a lot of these cubes. They will be used to test the gravity and collision detection parts of the assignment. If the viewpoint moves off of the top of one of these cubes then gravity should cause it to fall to the ground. The user should not be able to move to a position where they can see inside of these cubes or any other cubes in the game.

As the entire game world is on a two dimensional plane it might be useful to create a two dimensional array which represents the positions for rooms, doors, and corridors. This might be helpful when you are creating the corridors between rooms. Once the two dimensional array is complete you can use it to create the 3D game world. This isn't required but it can be easier to organize the world in 2D then build the 3D version using the 2D as a guide.

You can organize the world space in a 3x3 grid so the rooms cannot overlap but the dimensions and the position of the rooms must be randomized. A room may be constrained to appear within a space in the grid but it should not always start in the same place or have the same size. Room sizes should range from small to large.

2. Collision Detection

Add collision detection so the viewpoint cannot pass into a space which is occupied by a cube. Any world space that is not set equal to 0 is occupied.

Test for collisions by checking if the position the viewpoint will move into is an occupied space in the world. Write the collision detection so that the viewpoint does not pass inside a cube at any time.

An exception to this allows the player to climb on top of a cube. This occurs when the viewpoint would pass into a cube that is occupied but when there is nothing above that cube. In this case the viewpoint will move on top of the existing cube. This lets the player climb on top of something that is one cube tall. If there were two cubes stacked on top of each other then the player would not automatically climb on top of them and they would collide with the cubes normally (and stop moving).

The simplest method for collision detection is to test if the next move of the viewpoint will enter an occupied cube. If the cube is occupied then prevent the movement. This can be done using the get and set viewposition functions.

Responding to a collision when it is just about to occur may not always be sufficient. It is possible in some cases for the viewpoint to pass into the edge of a cube and not register as being inside the cube. This allows the user to see inside the cube when they should not be able to do so. You may need to take into consideration the direction which the viewpoint is moving and predict when it is getting near to the cube instead of it being inside the cube. In this case you are actually looking for the viewpoint being close to an occupied space instead of inside that space.

Add collision detection so the viewpoint cannot move outside the space defined by the world array. The viewpoint should not move to a position less than 0 in any direction. It should not move to a position greater than 99 in the x and z direction, and not greater than 49 in the y direction.

You can use the functions:

getViewPosition()
setViewPosition()
getOldtViewPosition()
setOldViewPosition()

to prevent the viewpoint from entering an occupied cube. You will need to test that the next viewpoint position will not be inside a space which is occupied in the world[][][] array (it cannot be equal to anything other than 0). If the next viewpoint position is occupied then you will need to set the new position to an older position which was not occupied.

A tricky part of the assignment will be preventing the viewpoint from entering the edge of a cube. It is fairly easy to stop the viewpoint from passing through a cube but it is more difficult to stop it from peeking inside the edge of a cube. It is not impossible but it may take some experimentation with when controlling the viewpoint.

3. Gravity

Add gravity to the program so the viewpoint descends to the ground.
The rate of descent should not be too fast (e.g. 0.1 per update).
Note that because the indices are negative, gravity is an increase in y.

Gravity operates like a collision with the ground. If gravity would push the viewpoint into an occupied cube then it should not be be allowed to move the viewpoint.

The user should not have to press f for gravity to take effect at the start of the game. It should be in effect once the game starts.

4. A Few Things to Think About

Timing of events will be added in a later assignment. This will require the update() function to be modified so that events occur on a times schedule and are not dependent on the speed of the processor. You don't need to include this with assignment 1 but you can consider it when modifying the update() function. Events will eventually timed to they occur on a schedule and not every time the update() function is called.

This may not be a problem you encounter with this assignment but if you are having problems with the old and new positions not being the same when there is no keypress then you need to think about how the positions are updated by the system. The system responds to two types of events. Either the player presses a key (a keyboard() event) or it operates when nothing else is happening (the update() function). The keyboard function will set the old and new view positions to be correct after a keypress. The update function doesn't automatically update the old position so it will never change. If no key is pressed then the position variables aren't updated and they contain the movement values from the last time a key was pressed. It is up to you to make sure the position variables are modified in update.

Viewpoint Motion
By default the system moves in the direction of the w, a, s, or d key when pressed. When one of these keys is not being pressed then the motion stops.

Choosing Parameters
It is important to pick values for parameters such as colours, speed of objects, the effect of gravity so they are easy for the marker to see.
If the effect of a parameter it isn't obvious or is difficult to see then it will be marked as missing or incomplete.

Make sure colours are distinct. Choose velocities for moving objects
that are fast enough to be seen.

Coding Practices
Write the code using standard stylistic practices. Use functions,
reasonable variable names, and consistent indentation.
If the code is difficult for the TA to understand then you
will lose marks.

As usual, keep backups of your work using source control software.

Starting Code
The starting code is available on the Courselink site.
You can untar the file using tar xf filename.
All of the changes to the code can be made in the a1.c file.

Note that the graphics code may be modified for assignment 2. If you
make changes to the graphics code (in graphics.c or visible.c) then you
may have to recode the changes in assignment 2.

Submitting the Assignment
Put all of the files in a directory names 4820 so they unpack into
this directory.

Submit the assignment using Courselink. Submit only the source code,
the makefile, and any documentation. Bundle the code in a tar file.

Include a makefile that will compile the executable. Name the executable a1.

The TA will unpack your code and type "make". They will then try to
run an executable named "./a1". If the make command or executing a1
does not work then you will lose a substantial number of marks.

Don't name your program with a .exe extension. If the assignment
says name the executable a1 then don't change the name to a1.exe.

It is always a good idea to unpack and test the file you are submitting
to be sure that what you submit actually compiles.