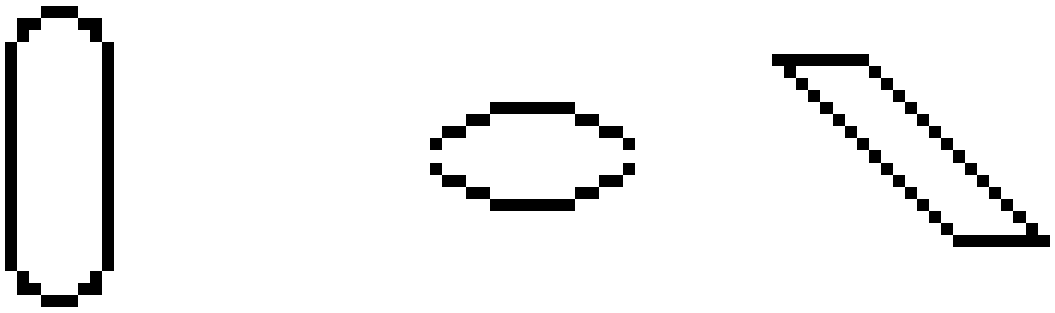
ECE3090 **Spring Semester, 2012**

Project 7 - Path Discovery and Obstacle Avoidance

Assigned: Apr 5, 2012 *Due: Apr 22, 2012, 11:59pm,*

Introduction. The objective of this assignment is to navigate an avatar from a starting position to an ending position in an X–Y plane, and avoiding obstacles along the way. The X–Y plane with obstacles is given as a grayscale image in file `obst.png`. There are three different avatar images, a capsule shape `capsule.png`, a lens shape `lens.png`, and a rhombus shape `rhombus.png`. The obstacles image is 256x256 pixels, and each of the avatars are 32x32. All images are 8-bit gray scale. In the obstacles image, a white pixel is not obstructed, and a non–white pixel represents an obstacle.

Your avatar navigates from a specified starting position to a specified ending position. The avatar can move one pixel in any direction, including diagonally. In addition, the avatar can *rotate* about it's midpoint. For simplicity, we will assume rotations in units of 10 degree per rotation. In other words, each avatar has only 36 possible rotated orientations, representing rotations of 0 to 350 degrees. A rotation of +10 degrees results in a *counter–clockwise* movement of the avatar.



Getting Started. First, the skeleton files are on `jinx` in the usual location, subdirectory `PathExploration`. To implement the algorithm you will need to read in the two image files, one for the obstacles image and one for the avatar image, and process those as a one–dimensional array of pixels (bytes), identical to the pixel arrays we have discussed in class often. The API to `QDisplay.h` is identical to what we used previously.

Next you will need a `RotateImage()` subroutine. Given an image, it will rotate it about the center point by the specified number of degrees. If the image has an even number of pixels in the width/height, there is not a true "center" pixel, so in this case just rotate about the pixel at (Width / 2, Height / 2). A separate pixel buffer for the input (non–rotated) and output (rotated) image works best, but not a requirement. Since all of the math to do the rotations is in *floating point*, we should round the results when converting back to integer pixels (by adding 0.5 before the cast back to an `int`, or calling the `round()` function).

You will need a `CheckValid()` subroutine to test if a given avatar with a given rotation value can legally be placed on the obstacle image at a specified location without "bumping into" an obstacle. Whatever method you use, keep in mind that this test will be used many times in the search algorithm, so something efficient is needed. **IMPORTANT.** The specified X–Y position of the avatar placement is relative to the *center point* of the avatar. Since our avatars are 32x32 images, the avatar extends 16 pixels from the center point in the up/down/left/right directions.

You should write and test the `RotateImage()` and `CheckValid()` routines separately, and be confident they are correct before proceeding with the rest of the assignment.

The Search Algorithm. The problem of finding a path from a starting point to an ending point has been studied for decades, and is well understood. However, in our instance it is complicated by the fact that the avatars can be rotated, leading to 36 different avatar images that might be placed at each point. One way to deal with this multiple–avatar problem is to make the X–Y plane into a three–dimensional X–Y–Z plane, where the Z axis represents the rotation angle. In other words, for our path calculations we use a 256x256x36 cube of possible locations. As the algorithm is searching for a path, each iteration tests 26 possible one unit moves (one unit in each of the X–Y–Z directions including diagonals). Keep in mind that the Z axis is circular, meaning that moving down in the Z direction from Z = 0 moves to Z = 350, and moving up from Z = 350 moves to Z = 0. For the remainder of this document, we will refer to a location in the 3–D cube as a *cube element*.

In our approach, each move has an associated *cost*. To keep things simple, we will have only three possible costs; either $1, \sqrt{2}, \text{or} \sqrt{3}$. A cost of 1 is used when we move only one unit in one direction only (for example, directly left). A cost of $\sqrt{2}$ is used when we move diagonally (for example, left and down). A cost of $\sqrt{3}$ is used when we move diagonally along with a rotation (for example, left, down and 10 degree rotation in either direction).

The algorithm starts at the destination point and works backwards towards the starting point, keeping up with how far from the destination each possible point is. The algorithm terminates when all valid positions have been checked, or when the starting point is reached. The algorithm we are using is a variation on the famous *Dijkstra's algorithm*, which finds the shortest path from a source to a destination in a weighted, directed graph.

The variables needed for the algorithm are:

1. A `distance` vector (or array) that indicates how far each cube element is from the specified destination. Initially, all values are infinity, excepting the destination itself, which is of course 0 units from the destination.

2. A `pending` container, which holds the cube elements for which we know a distance, but that distance may not yet be the shortest. The contents of the container are an object that contains a cube element location, and a *distance* value indicating how far that element is from the destination cube element. This container ***MUST*** be sorted, in ascending order of distance from the destination. The initial contents of the `pending` container is only the destination itself, with (of course) a distance of zero. You will likely want to use a Standard Template Library `multimap` for this container.

3. For an efficient implementation, you may want some other arrays or vectors to keep up with known information about each cube element.

The algorithm is as follows:

1. Initialize the `pending` container with a single entry indicating the destination pixel point and a cost of zero to reach the destination from that point.

2. While the `pending` container is *Not Empty*,

   (a) Get the first (closest) element from the `pending` container and remove it from the container. Define `thisDist` as the distance from the destination for this element.

   (b) For each neighbor of this element, first determine if that location is a valid avatar placement (ie. it does not bump into an obstacle). If it is not valid, ignore it.

   (c) Define `moveDist[i]` as the cost for a move to each of the 26 possible neighbors. This is either $1, \sqrt{2}, \text{or} \sqrt{3}$ as discussed previously.

   (d) For each valid neighbor of this element, calculate the distance from the distination for that neighbor as follows:

   `neighborDist = thisDist + moveDist[i].`

   (e) If `neighborDist` is *less than* the distance for that neighbor found in the global `distance` array, we have found a potential shorter path, and perform the following. If not, ignore it.

      i. Set the new shorter distance to this neighbor in the global `distance` array.

      ii. Add this neighbor to the `pending` container. Remember that this container is *sorted*, using the distance as the sort key. This will cause all neighbors of this one to be checked later, checking closest ones first. Keep in mind that this neighbor might already be in the `pending` container with a longer distance.

      iii. If the current element is the starting location, the pending loop terminates. Unless the starting location is *not valid*, this should always occur *before* the `pending` container goes empty.

   (f) At this point, the algorithm has terminated and completely populated the `distance` container, with each element containing the explored cube elements and their shortest distance to the destination. The value stored in `distance` for the starting point cube element is the minimum distance to the destination. There are of course likely to be many pixels that were not "visited", but there are known to be not of interest as they are not along the shortest path.

Animating the Results. Once your algorithm has found the shortest path, we need to create an animation of the path. Start by displaying the obstacle course background, with the avatar placed at the starting location (with the specified rotation). For each of the 26 neighbors of this point, find the one *closest* to the destination (the distance to the destination is in the `distance` vector used during the algorithm) and move the avatar to the closest one, again with the correct rotation. Be sure to erase the avatar from the previous location. Continue finding the closest neighbor of the current position until the destination is reached.

Further, we must minimize screen flicker. Being lazy and simply calling `Update()` whenever the image changes is not the right approach and will cause excess flickering. You should keep up with what portions of the screen have changed on each update, and use the `Update()` call that specifies a partial screen update. This will result in smooth animation. Use an animation update rate of 24 frames per second as specified on the `UpdateRate` method of `QDisplay` objects. You can experiment with faster update rates.

Testing your program. The images, starting x/y/z position and ending x/y/z position are passed to the program as command line arguments. An example would be:

```
./findpath obst.png capsule.png 54 85 60 124 246 270
```

The above says to use the capsule avatar, start at $x = 54, y = 85$ and a 60 degree rotation; and find a path to $x = 124, y = 246$, with a final 270 degree rotation.

A set of 10 sample starting and ending positions, along with the correct path for each will be provided.

Grading. The program will be graded as follows:

1. Finding the correct path for each of the 10 samples. **8 points each**. There may be multiple paths with the same distance, but the distance along the path must match the given results.
2. Path animation **1 points each**
3. Flicker free updates during animation **1 points each**.

Turning in your Project. Use the usual `riley-turnin` procedure to turn in your project.