CS1112 Fall 2012 Project 6 Part B due Thursday 11/29 at 11pm

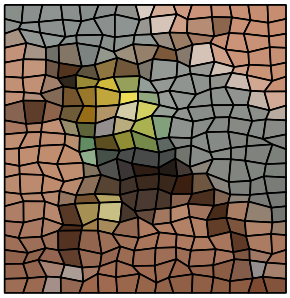
You must work either on your own or with one partner. If you work with a partner you must first register as a group in CMS and then submit your work as a group. *Adhere to the Code of Academic Integrity.* For a group, "you" below refers to "your group." You may discuss background issues and general strategies with others and seek help from the course staff, but the work that you submit must be your own. In particular, you may discuss general ideas with others but you may not work out the detailed solutions with others. It is not OK for you to see or hear another student's code and it is certainly not OK to copy code from another person or from published/Internet sources. If you feel that you cannot complete the assignment on you own, seek help from the course staff.

Objectives

Completing this project will solidify your understanding of structs and struct arrays (Part A), object-oriented programming (Part B), and recursion (Part C).

Part A (question 1) appears in a separate document.

2 Stained Glass Production, Take 2



In Project 4, you applied a "stained glass effect" to a jpeg file using procedural programming; this time you will use object-oriented programming (OOP)! *Before* starting on this part of the project, *review* the posted solution to the stained glass problem in Project 4.

2.0 Object-Oriented Design

We provide the *design* of the classes that you will *implement* in this project. Read carefully about the reasoning behind the design below.

Look at the example stained glass image. How can we characterize that image (and other stained glass images)? We see a set of *tiles*, or you may say a set of *grid points*. The grid points are at first uniformly distributed, but the interior ones are then "perturbed" in order to make interesting looking quadrilateral tiles. Given these observations, it may be reasonable to have a `StainedGlassImage` class that has these properties: the image data, the number of rows and columns of tiles, the set (the array) of tiles, the set (the array) of grid points, and a value to control the amount of perturbation to the grid points.

Each *tile* has essential properties: a color and the four "bounding" *vertices*. Each *vertex* is really a grid point and it also has intrinsic properties: the x and y coordinates, and in the context of this problem, a differentiation between an edge vertex and an interior vertex.

We have identified three kinds of objects—the vertex, the tile, and the stained glass image—so we can define them in three classes: `Vertex`, `Tile`, and `StainedGlassImage`. Their properties were discussed above, but how about the operations that should be performed on them, or that they should be able to perform?

`StainedGlassImage`

The "stained glass effect" is achieved by breaking an image into a set of tiles, so this operation—computing the grid points and therefore identifying the tiles—should be defined in class `StainedGlassImage`. Of course, the class should also show the image with the stained glass effect applied. *Read the skeleton (incomplete) file* `StainedGlassImage.m` *now to see the properties and methods that belong to the definition of class* `StainedGlassImage`.

`Tile`

A `Tile` has the properties color and vertices. What should a `Tile` be able to do? Not much ... other than being able to show (draw) itself. *Read the skeleton file* `Tile` *now.*

`Vertex`

A `Vertex` has some starting x and y coordinates and is either an edge vertex or not. It needs to be able to perturb its coordinates if it's not an edge vertex. *Read the skeleton file* `Vertex` *now.*

You will complete the class definitions by filling in the methods and adding extra methods. *Do not change the design, specifications, names (of classes, properties, methods, and parameters), attributes, and function headers given.*

2.1 Class `Vertex`

A `Vertex` is really a pixel location from the image, specifically, a `Vertex`'s x-coordinate can be represented by the pixel column number from the image. A `Vertex`'s y-coordinate can be represented by the *negative* pixel row number from the image. (Recall that pixel row numbers of an image increase going *down* while the y-coordinate in the Cartesian plane increases going *up*; hence the negative sign when relating these two values.) Notice that this way of representing the x and y coordinates of a pixel is simpler than what we did in Project 4!

Implement the constructor and method `perturb` according to the given specifications (function comments). Do you *need* getters and setter methods for the properties in `Vertex`? Do not add getters and setters for now. If you later discover that you need them, then add only those getters (and setters) that are *strictly necessary.*

Testing

Test this class, both to make sure that it is correct and to make sure that you know how to use it, before moving on to the other classes. Here are some suggested tests that you can do from the *Command Window*:

1. Create an edge `Vertex`, e.g., pixel at row 1 column 50, and use any value for parameter `maxNoise`:
   `v = Vertex(1,50,1,15)`
   MATLAB will say that you have a `Vertex` with *no* properties; this is because the properties' access attribute is `private`. (So MATLAB actually means that there are no *visible* properties.) Go back to the class definition and add getter methods for the properties `x` and `y`. Then re-do the test:
   ```
   v = Vertex(1,50,1,15);
   v.getX(); % You should see the value 50 displayed since no perturbation is done
   v.getY(); % You should see the value -1 displayed since no perturbation is done
   ```

2. Create an interior `Vertex`, e.g., pixel at row 120 column 40, and use 15 for `maxNoise`:
   ```
   w = Vertex(120,40,0,15);
   w.getX(); % There should be perturbation so you should see a value in (25,55)
   w.getY(); % There should be perturbation so you should see a value in (-135,-105)
   ```
   Try another interior vertex and see if the x and y coordinates make sense.

3. This test you should write in a script file so that you can easily copy and paste code and run the test multiple times (assuming that you may need to do some debugging). Call this script file `test.m`; write in it the following code to create an array of `Vertex`s:
   ```
   A(1) = Vertex(300,50,1,15);
   A(1).getX() % does the displayed value make sense?
   A(2) = Vertex(120,100,0,15);
   A(2).getY() % does the displayed value make sense?
   % For the next Vertex, skip to component 4 (leave out component 3):
   A(4) = Vertex(120,40,0,15); % if you get an error, then likely it is
                               % because your constructor does not handle
                               % the case where no argument is passed
   ```
   Run your script `test`.

Did all of the above tests work? *Do you understand the syntax?* If your answer is "no" to either question then *go to a course staff to get help!*

2.2 Class `Tile`

Implement the constructor as specified and start testing. (You don't have to implement method `draw` yet because this constructor doesn't need `draw`.) Modify your script file `test.m`:

1. Create a length 4 array of `Vertex`s called `A` with four real `Vertex`s—no empty components in the array this time.
2. Create a red `Tile` called `t` with the four `Vertex`s referenced by array `A`:
   `t = Tile([1 0 0], A)`

Run your script `test`. MATLAB should display text saying that you have a `Tile` with no (visible) properties. As long as there is no error message, it is good enough for now. We will do more testing after implementing method `draw`.

Implement method `draw` by making use of the provided function `myFill`. Forgot how `myFill` works? Read its function comments and look back at the solution to Project 4.

Testing

Continue to develop the script file `test.m` in order to do more testing ...

1. At the top of the file add these commands:
   ```
   close all % close all figure windows
   hold on
   ```
2. Keep the existing code that creates a red `Tile` referenced by `t`.
3. Call `Tile t`'s `draw` method:
   ```
   t.draw() % A figure window will open automatically to show the Tile
   ```
4. Create another `Tile` and call it `u`.
5. Type the following to show `Tile u` in the same figure window as `t`:
   ```
   u.draw()
   hold off
   ```

Run your script `test`. Did you see your two `Tile`s? Make sure you have a working `Tile` class before moving on to class `StainedGlassImage`!

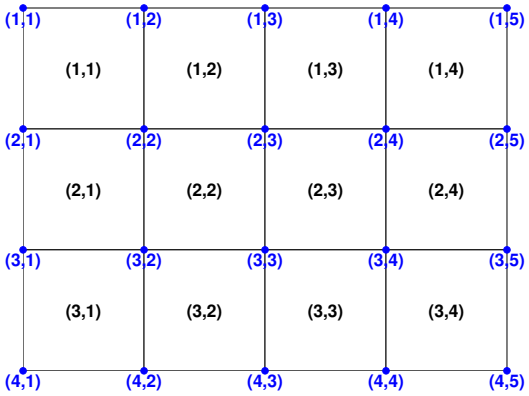
2.3 Class `StainedGlassImage`

Read the class *comments* near the top of the file and the properties comments. Read the provided constructor.

2.3.1 Method `applyEffect`

This is where you write the code to apply the stained glass effect:

- Determine the beginning pixel row number for each tile.
- Determine the beginning pixel column number for each tile.
- Create the set of grid points—the 2-d array of `Vertex`s. The figure below shows an example numbering of the grid points and the tiles. The grid points are numbered in blue; the tiles are numbered in black.



Let the grid points be the top-left pixels of each tile except for the last row and last column of grid points. The last column of grid points can correspond to the right edge of the rightmost tiles; the last row of grid points can correspond to the bottom edge of the bottom row of tiles.

In order to create a `Vertex`, you need to specify the amount of perturbation (argument to `maxNoise` in the `Vertex` constructor). Use this formulation: let d be the smaller dimension of a tile, i.e., the smaller value between the number of rows of pixels and the number of columns of pixels in a tile. An approximate value is fine since all the tiles may not be exactly the same size. Let `maxNoise` be $f \times d$ where f is the value in property `frac` of `StainedGlassImage`.

- Create the set of (the 2-d array of) `Tiles`. Be sure to give each `Tile` the correct four `Vertexs` from the set of `Vertexs` created in the previous step. Each `Tile`'s color is the average color of the block of pixels belonging to the tile—average the red, green, and blue layers of data independently. The `Tile`'s color should be a length 3 array storing rgb values; each value is of type `double` and is in the range [0,1].

To test your `applyEffect` method, simply attempt to create a `StainedGlassImage` object since the constructor calls method `applyEffect`. Since you have not implemented method `showEffect` yet, in the constructor comment out (i.e., put a comment symbol, the percent sign, in front of) the call to method `showEffect` for now. Then in the *Command Window*, type

```
SGI = StainedGlassImage('photo.jpg', 30, 20, .35)
  % First argument is the name of the jpeg file
  % 2nd and 3rd arguments are the number of rows and number of cols of tiles
  % Last argument is the fraction for perturbing interior grid points
```

Any error message? Since the properties of this class have `public` access, you can display their values, e.g.,

```
SGI.nr                          % You should see 30, the 2nd argument above
[nr, nc, np] = size(SGI.imData) % The numbers of rows, columns and layers of image data
SGI.tileSet                     % Matlab tells you it is a 30x20 array of Tile handles
```

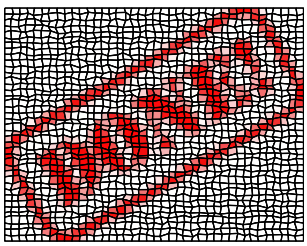
If there are errors, start debugging and ask for help from course staff if necessary.

2.3.2 Method `showEffect`

This method shows the final result—the stained glass image. Since the stained glass image is a set of `Tiles`, simply have each `Tile` call its own `draw` method! Use the typical graphics setup commands:

```
figure
axis equal off
hold on
```

and use `hold off` after the drawing is done. After implementing this method, in the constructor *uncomment* the call to `showEffect`. Now you should be able to create your `StainedGlassImage` object and see the image!



Submit your files `Vertex.m`, `Tile.m`, and `StainedGlassImage.m` on CMS.