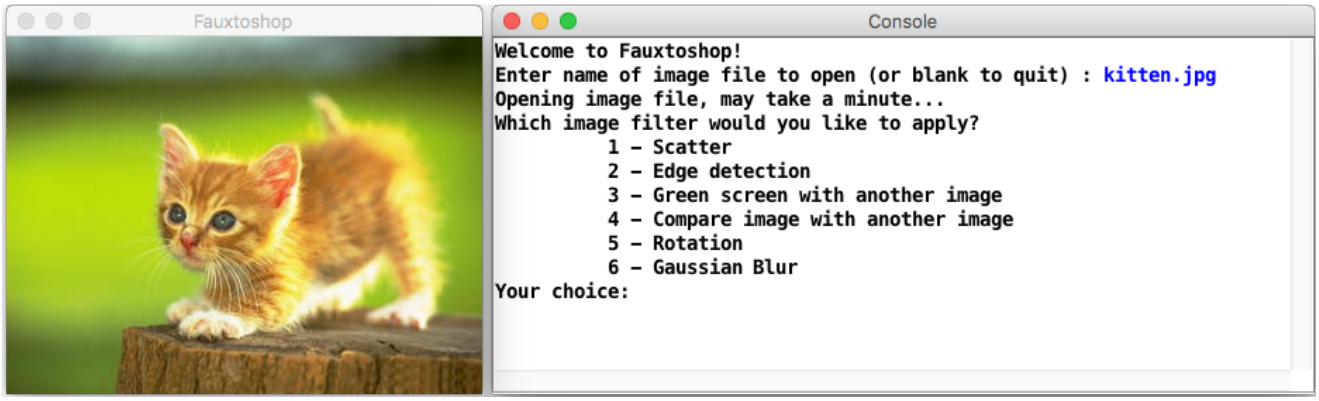
# Assignment 1: Fauxtoshop

Assignment by Cynthia Lee; inspired by Mark Guzdial and Barbara Ericson; name by Eric Yurko and Arun Debray.

**JANUARY 11, 2017**




## Due Date:

**Fauxtoshop is due Friday, January 20th at 12:00pm.**

## Assignment Overview

In this assignment, you'll write your own **photo editor** with advanced features like **green screen**, **"scatter"** blur effect, **edge detection**, **rotation**, and a very interesting type of blur called a **Gaussian Blur**. You'll be making extensive use of our week-1 topics: console interaction and the Grid ADT (look for our other ADTs to show up on assignment 2). For extra spice, there is one easy mouse click interaction. This document is long, but don't let that overwhelm you! (Much of it is just screenshots anyway.) You might even have some extra time, in which case there are plenty of opportunities for artistic expression and creative extensions (extra credit). We look forward to seeing what you do!

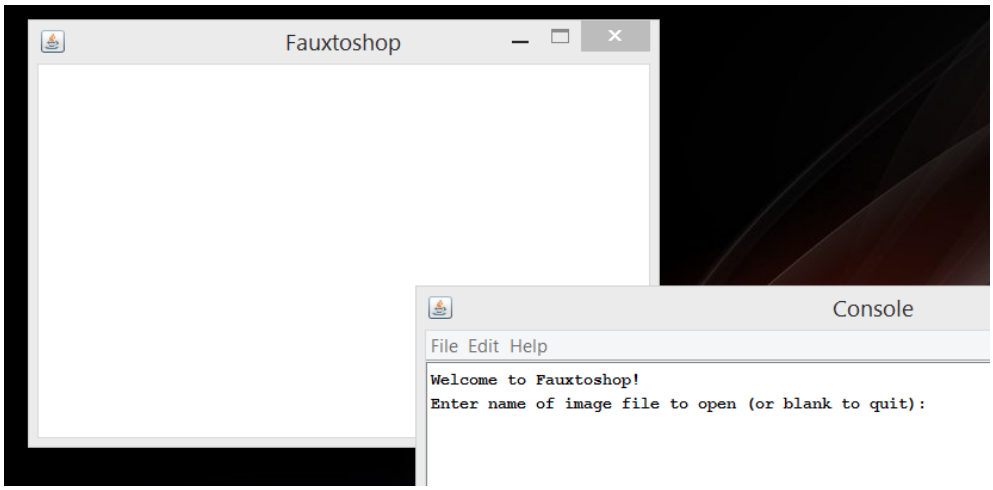
The assignment has several purposes:

1. To stress the notion of problem decomposition and taking a thoughtful, organized approach to program function and style.
2. To become more familiar with C++ strings and console I/O.
3. To gain practice with the Grid data structure.
4. To evaluate use cases for pass-by-reference parameters vs pass-by-value parameters, and when to use `const`.

*Note: CS106X does not have pair programming, and all assignments should be done on your own.*

## Console Window and Graphics Window

Your program should use the console to support basic menu selections, file name input, and other interactions with the user. Below is a screenshot of the initial welcome screens:




You'll notice there are **two windows**: Console and a Fauxtoshop window. In the screenshots throughout this document, you will see that what you should print in the window labeled "Console" is in black font, and what the user types is in blue font. The "Fauxtoshop" window is created using the Stanford library graphics class **GWindow** (refer to Stanford library documentation). *Very important note*: **your program should only ever declare and use one GWindow**, and its lifespan should span the entire duration of the program, or else your program may crash. We have declared one in main for you in the starter code, thereby making its lifespan the duration of the program. You should not edit the main code or make other GWindow objects.

# Main Menu Sequence

This section outlines the sequence of interactions that take place in the main menu in the console window. Later sections will explain other important components of the program, and detail the behavior of each of the four menu options.

First, greet the user with the welcome message provided, then ask the user to specify an **image file name**. The image files should be located in the **"res" directory** (sibling directory of your "src" and "lib" directories in the project directory). When images are located there, QT will handle making them available to your code without additional path (directory location) information.

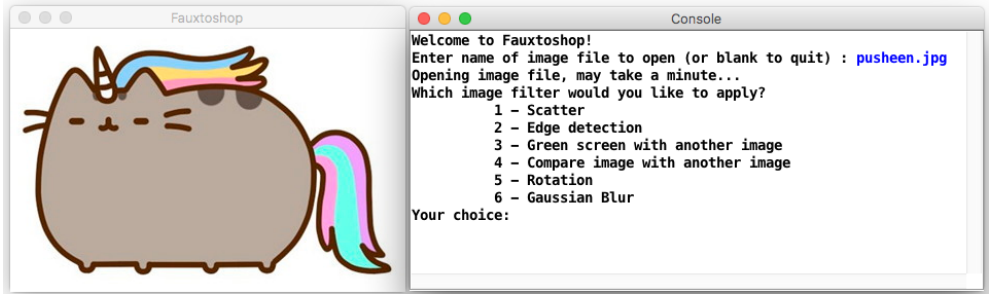
To open the image file, declare a GBufferedImage object, and then call the openImageFromFilename function (provided in the starter code) to open the file with the image file name the user specified. This function returns a Boolean value, where true indicates success. You should re-prompt the user for a filename if this function returns false. After the image is opened, you'll want to resize the GWindow to be the exact same size as the image, and then add the image to the GWindow by calling add (the code shown below assumes your GBufferedImage object is called img and your Gwindow object is called gw, but you could use different names):

```
gw.setCanvasSize(img.getWidth(), img.getHeight());
gw.add(&img,0,0);
// You'll notice there is an & before img. This is not the same kind of &
// reference parameters. We'll talk about it later in the quarter. For no
// copy this line of code when adding images to a GWindow.
```

**It is important that the GBufferedImage object you add to the GWindow not cease to exist (go out of scope) after you add it to the GWindow**. For example, if you make a helper function that declares a GBufferedImage object and adds it to the GWindow, then returns (causing the local variable of the GBufferedImage to go out of scope and cease to exist), this could cause your program to crash! The starter code declares one GBufferedImage in main, and if you use that object throughout the program (by changing the contents, not declaring a new one to add to the GWindow—though you may declare temporary ones that you don't add to the GWindow), you'll avoid any trouble. Another way to avoid trouble is that if you call gw.clear(), it's ok for any GBufferedImage objects that were added to the window to cease to exist, because the GWindow has then "forgotten" about them.

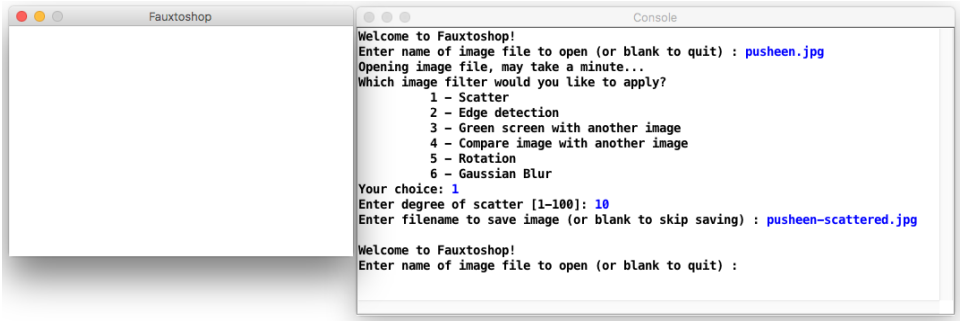
Below is a screenshot showing the next steps of the interaction after the image is opened and added to the window. It shows a menu of filter effect options and asks the user to select one (if the user enters something other than a valid menu choice, reprompt with the entire menu text starting with "Which image filter…"). Each menu option has additional input(s) required of the user, which are explained in each menu option section below. By the way: some of the examples shown in

this document may take a very long time to render, depending on the speed of your computer. When opening larger image files than your debugger can handle well, it may appear that the program has frozen. If this happens to you, try working with smaller images.




After the filter effect is complete and displayed in the window, you should ask the user if they want to **save the image**, as shown in the screenshot below. To save the image file, call the saveImageToFilename function (provided in the starter code) with the image file name the user specified. This function returns a Boolean value, where true indicates success. You should re-prompt the user for a file name if this function returns false. The saved image file will not save in the input images' "res" directory, where you might expect! Instead, it will save in the build directory created by the compiler to hold its intermediate work and the final compiled executable. The build directory will be found as a sibling directory of the "Fauxtoshop" project directory, and it will have a name something like "build-Fauxtoshop-Desktop_Qt_5_7_0_clang_64bit-Debug". Usually you should ignore the build directory and not mess with it, but it's fine to look in there for your saved image files.

After offering to save the image, clear the GWindow object so it is blank again (call its clear method). Print one blank line. Then repeat the entire main menu sequence again (as shown in the screenshot below), until the user enters an empty filename to quit the program.




# Working with Images

For each of the filter effects in the menu options, you'll want to iterate over the pixels of the image and inspect or change them. Before doing any such inspection or change of pixels, convert your GBufferedImage object into a Grid format (yay, ADTs!). The conversion step is as follows (where img is the name of the GBufferedImage object; yours could have a different name):

```
Grid<int> original = img.toGrid();
```

The individual pixels (row/col entries) of the Grid are integers representing **RGB** values. You can read about the RGB color scheme in detail on Wikipedia or RGBWorld, but the basic idea is that there are three separate values packed into one int: a value between 0 and 255 representing how much **Red**, a value between 0 and 255 representing how much **Green**, and a value between 0 and 255 representing how much **Blue**. The red, green, and blue combine together to make one particular pixel's color. By varying the red, green, and blue values, we can make all the colors of the rainbow. It's a slightly different version of the Primary Colors concept you learned as a kid. In the starter code, **three color values are defined for you** as constants:

```
const int    WHITE = 0xFFFFFF;
const int    BLACK = 0x000000;
const int    GREEN = 0x00FF00;
```

These numbers are written in base 16 (called "hexadecimal"), which you don't need to worry too much about. They still behave as any other const variable of type int. For the curious: The first two digits represent red, the second two digits represent green, and the third two digits represent blue. So in the case of white, the red, green, and blue components all

have the maximum value of 255 (base 10) or “FF” (base 16). On the other hand, green has the maximum value of green (the middle two digits are “FF”) and 0 of red and blue.

We have a nice helper function provided for you in the Stanford libraries that separates the pixel ints into their individual RGB components for you (it’s a nice example of “returning” three values using pass-by-reference!). Here is an example for one pixel (where original is the name of a Grid object, and row and col are integers; as usual, you may choose different names):

```
int pixel = original[row][col];
int red, green, blue;
GBufferedImage::getRedGreenBlue(pixel, red, green, blue);
```

The GBufferedImage::getRedGreenBlue syntax might look new or strange to you. The “::” is called the scope resolution operator, and it indicates that the getRedGreenBlue function can be found in the GBufferedImage class, but that you do not need an instance of the class to use the function (that is, it is a static function in the same sense of the word static that you may have seen in Java). You don’t really need to worry about exactly what this means until we talk about it later in the course. Just use the function as if its "full name" is GBufferedImage::getRedGreenBlue.

You should ***not*** **use the GBufferedImage methods such as setColor or setRGB that set individual pixels in the image**. Instead, make all the changes you want in a Grid object, and then change the image all at once by calling the img.fromGrid (where img is the name of your GBufferedImage instance), which takes a Grid as its argument. There are two reasons for this requirement: one is efficiency, and the other is to get you to practice using the Grid ADT by doing looping and operations there rather than in the image object directly.

# RGB Color “Difference” Calculation

For both the edge detection and green screen effects, your program will need to have a way of calculating the “difference” between two colors. There are a number of ways to do this, and in fact exploring alternate calculations could be one of the extensions you do for this assignment.
To earn points for the basic assignment you’ll need to precisely implement our algorithm (and not some other algorithm, however reasonable).

1. Use the GBufferedImage::getRedGreenBlue method described above to separate the pixels into their RGB components.
2. Take the difference between each pair of RGB values (difference between the two reds, difference between the two blues, and difference between the two greens)
3. Take the absolute values of each difference so you have the magnitude of each difference. Note the standard library has an integer abs() function.
4. Then take the max of those three differences (there is also a max() function that takes two values at a time).

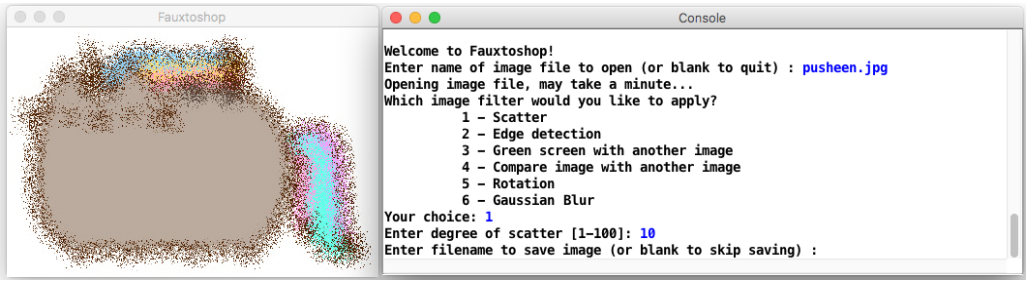
That is going to be our definition of difference between pixels. *Note, all calculations should be done using type int, not double/float.*

# Menu Option 1: Scatter

For this filter, your program will take the original image and “scatter” its pixels, making something that looks like a sand drawing that was shaken.
First, ask the user to provide a “degree of scatter” for how far we should scatter pixels. The value should be an integer between 1 and 100, inclusive (otherwise reprompt).

Then create a new image Grid that has the same dimensions as the original image Grid. Next, for each pixel in the new image, randomly select a pixel from nearby that row and column in the original image that will provide the color for this pixel in the new image. You will randomly select the pixel by randomly selecting a row that is within radius of the current row, and randomly selecting a column that is within radius of the current column. If the randomly selected row or column is out of bounds of the Grid of the original image, you should randomly select again until you get an in-bounds pixel.

Here is a screenshot of the Scatter filter effect complete, waiting for the user to respond to the prompt about saving the file (notice we haven’t cleared the window yet):

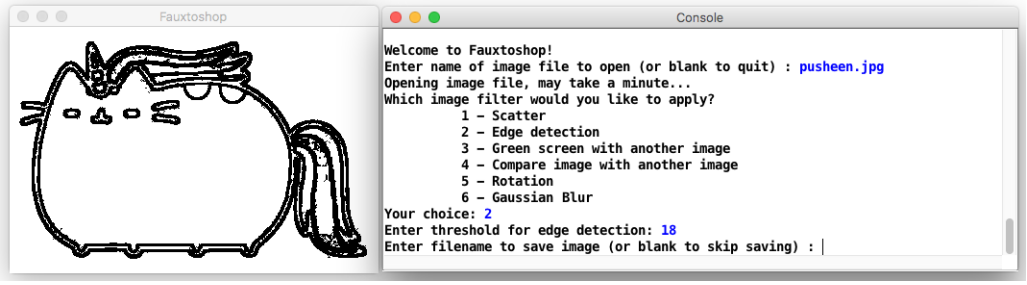



# Menu Option 2: Edge Detection

For this filter, your program will create a new black and white image of the same size as the original, where a given pixel is black if it was an edge in the original image, and white if it was not an edge in the original image.
First, ask the user for a **threshold** that controls how different two adjacent pixels must be from each other to be considered an "edge." This should be a positive (nonzero) integer value (otherwise re-prompt). Then, loop over each pixel and determine if it is an edge or not.

A pixel is defined as an **edge** if at least one of its neighbors has a difference of *greater than threshold* from it. The neighbors are the 9 pixels (including self) immediately adjacent or diagonal from the current "self" row/column of the Grid. So if my distances to my neighbors are: 9, 8, 5, 3, 3, 0 (self), 4, 7, 8, 7, then I would only be considered an edge if the threshold were less than 9 (the greatest difference between me and one of my neighbors). Remember that pixels near edges and corners may not have all 9 neighbors, so take care not to go out of bounds. The Grid class has an inBounds method that will be helpful.

Here is a screenshot of the Edge Detection filter effect complete, waiting for the user to respond to the prompt about saving the file (notice we haven't cleared the window yet):




# Menu Option 3: Green Screen

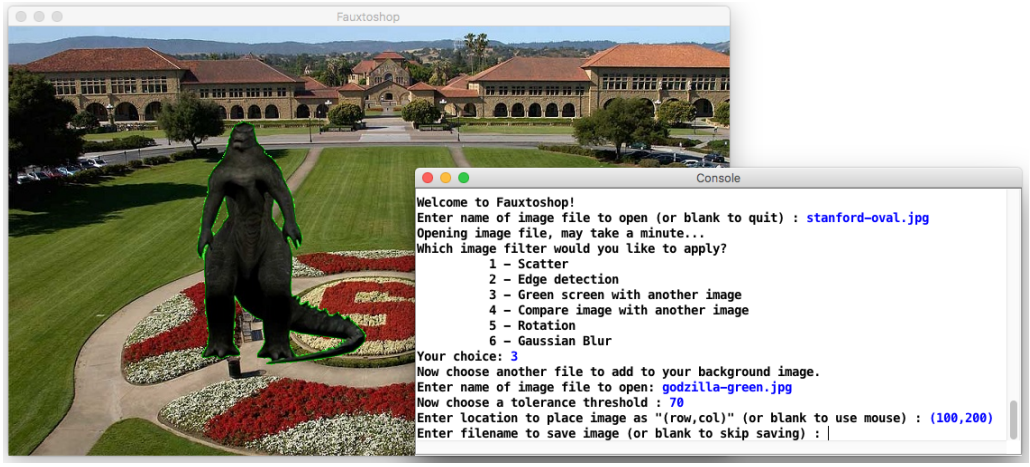
Menu Option 3: Green Screen For this filter, your program will paste a **"sticker" image** on top of a **"background" image**, but ignore any part of the sticker that is close to pure green in color. This technique is used widely in filmmaking—actors are filmed acting on a stage that is painted bright green and then their forms are later digitally placed on top of some other scenery.
The image that the user specifies in the main menu will be the **background image**, so the first thing you should do for this filter effect is ask the user to specify a new image file name to be the **sticker image**. The prompt will work in a similar fashion (reprompt if open fails), but you will not offer the option to enter a blank filename to indicate ending the program. Open the image file and convert it to a Grid in the same way you did with the original background image.

Next, we need to know **how much tolerance** we should have for green that isn't pure green (0x00FF00). The tolerance should be between 1 and 100. That way if there are slight shadows or other variations on the green part of the image, the green screen effect will still work as we want it to. Prompt the user to enter a value, and then you'll use that as a threshold in conjunction with the same pixel color difference calculation that we used in edge detection (described above).

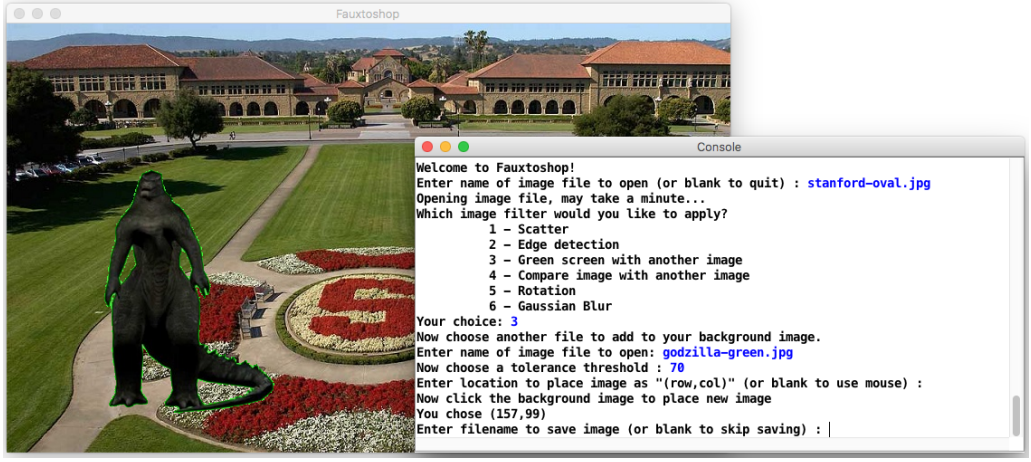
Next, we need to know **where to place the sticker** image. Ask the user to specify a location as (row,col) in exactly that format, with row and col being non-negative integers (otherwise reprompt). The row and column specified will be the background location where you will place the upper left corner of the sticker. The only variation from that exact (row,col) format that you should allow is for the user to enter nothing (just hits return), in which case you should allow the user to specify the location use a mouse click. The starter code includes a function that receives mouse clicks and reports the location to you as row and column. You should report back to the user the detected click location.

Now we are ready to **place the image** in the location specified. Any pixel on the sticker image that is difference *greater than* threshold from pure green will be copied onto the background, otherwise that pixel will be ignored and the background pixel there will remain untouched. You should allow the sticker image to be cut off on the bottom or right edge(s) if it cannot completely fit on the background.

Here is a screenshot of the Green Screen filter effect complete, waiting for the user to respond to the prompt about saving the file (notice we haven't cleared the window yet):

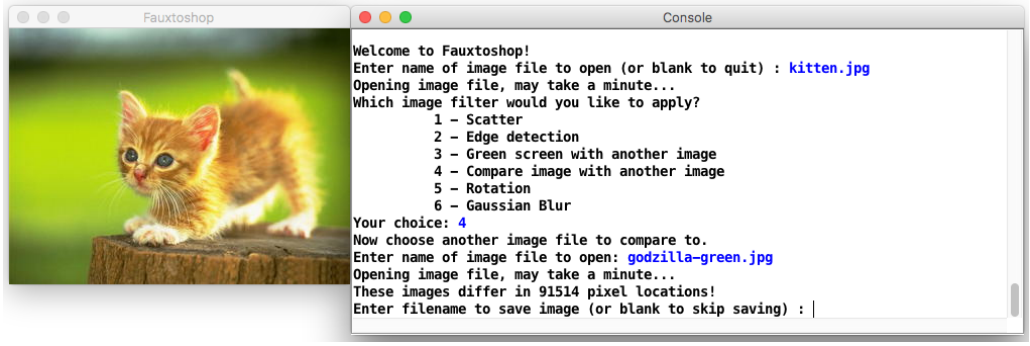



Here is a screenshot of the Green Screen filter effect complete in the case where the user clicked to give the sticker location:




# Menu Option 4: Compare Images

This is the least exciting of the menu options, and does not produce any visual effect. However, it will help you debug your code by comparing your output to the sample outputs provided on the course website. For this menu option, ask the user to name another image file (open it as a GBufferedImage in the same way described in Green Screen section), and you will count how many pixels differ between the two. This would be easy to do by iterating over the pixels yourself, but it's actually *even easier* than that—there is a countDiffPixels method in the GBufferedImage class that does all the work for you. Use it, then report the result to the user. Print a nonzero count as "These images differ in _ pixel locations!" or print "These images are the same!" as applicable. You will continue to display the original image. (It will be a bit superfluous when the main menu asks the user if they want to save the resulting image in this case, but that's okay.) Here is a screenshot showing this menu option in action:



# Menu Option 5: Rotation

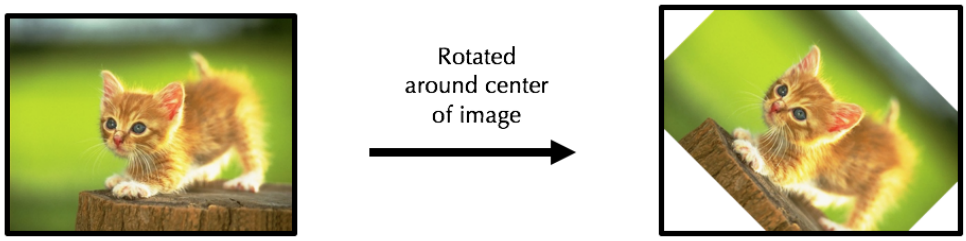
For this option, you will implement a rotation on the image. For this menu option, ask the user to enter an angle in degrees. Your program should then rotate the image clockwise for a positive angle, and counterclockwise for a negative angle.
One way to rotate an image clockwise around its center point is as follows: For each pixel in the rotated image at coordinate (x, y), copy the color of the pixel from the original image closest to (xOld, yOld) such that:

```
xOld = +x * cos(θ) + y * sin(θ)
yOld = -x * sin(θ) + y * cos(θ)
```

Where:

- $\theta$ is the angle of rotation.
- x coordinates represent number of pixels to the right of the image center (or, if negative, the number of pixels to the left of the center).
- y coordinates represent number of pixels below the image center (or, if negative, the number of pixels above the image center).

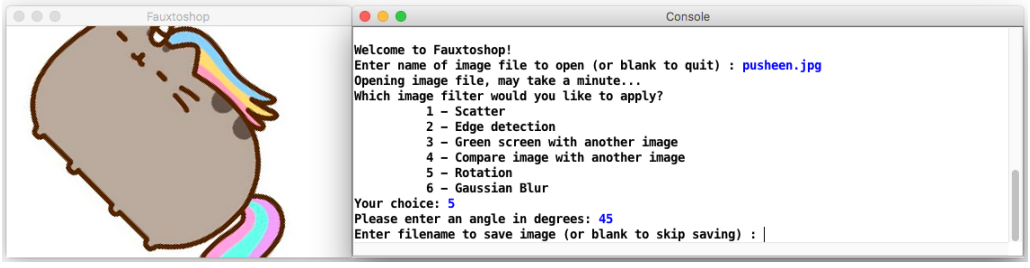
You will need to translate between rows and columns and x and y coordinates. As an example, the grid cell (row = height/2, col = width/2) translates to coordinate (x = 0, y = 0). One thing to consider when rotating a non-circular image is that some of the rotated pixels will not fit in the grid. Think about rotating the following rectangle by 45º:




Additionally, part of the new image will not receive any rotated pixels. In this case, we will make those pixels white (suggestion: make the new grid completely white to start).

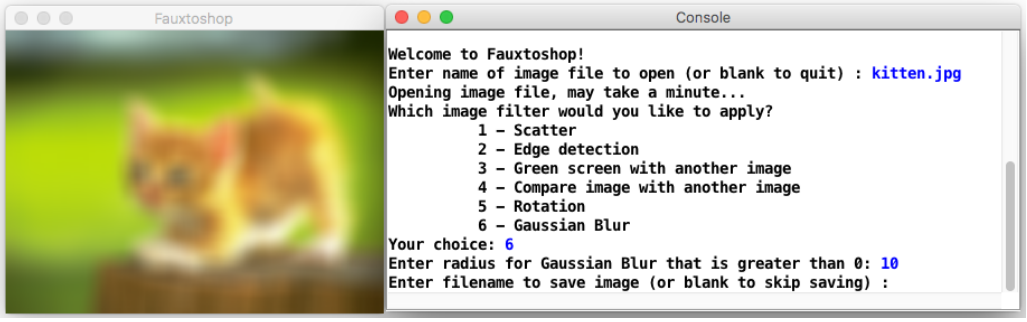
You will also need to make use of the **`"gmath.h"`** library to use the **`sinDegrees()`** and **`cosDegrees()`** functions. Alternatively, you can use the standard C++ **`<cmath>`** library, which defines the trigonometric functions using radians (but you will need to convert from degrees to radians if you use them).

Here is a screenshot of the rotation effect waiting for the user to respond to the prompt about saving the file (notice we haven't cleared the window yet):




# Menu Option 6: Gaussian Blur

The Gaussian Blur option is, arguably, the most difficult part of the project, and it is the one where we will give you the least amount of directions. A Gaussian Blur is a smoothing of the image using a "Gaussian function," and it produces a much smoother blur than scatter. The starter code does include a function that calculates a Gaussian Kernel for you, which is the trickiest part of the assignment. You should start researching what a Gaussian Blur is by going to the Wikipedia Article on Gaussian blur.
Here is a screenshot of the Gaussian Blur effect waiting for the user to respond to the prompt about saving the file (notice we haven't cleared the window yet):




# Grading and General Remarks

Please observe the following to avoid grade penalties:

- *Decomposition*: On this assignment part of your Style grade comes from making intelligent decisions about decomposing the problem into well-designed parts. Your functions should perform distinct and clearly delineated purposes, be clearly named, and not too long.
- *Parameters*: As much as possible, **pass collections and objects by reference, and const reference when applicable**, because passing them by value makes an expensive copy.
- *Collections*: Use the Grid and other Stanford library tools. Do not use pointers, arrays, or STL containers on this program. You should not be explicitly allocating or freeing any memory for this assignment (goes along with no pointers).
- *Style guide*: Refer to the CS106B+X style guide for further instructions.
- *Citation*: If you refer to any non-class resource (person, website, your own previous coding projects that are substantially similar), you need to make a clear citation of that fact in the code. Please refer to the Honor Code for CS106A+B+X for more detail.
- *Project size*: While high quality solutions can vary widely in length, some students find it helpful to know, as a general point of reference, how many lines of code our solution is. Our solution is about 350 lines of code, including #include, comments, etc.

# Extensions Suggestions

For all of these extensions, **make their use go through a new top level menu item**, so autograders can still test your program using menu items 1-4 and expect them to perform exactly according to specification. So, for example, if you change the behavior of green screen, make a new top level menu item 7 titled something like “Improved green screen.”Alternatively, you can submit a completely separate "extensionsCPP" files (no space) that the graders can use.
Edge detection extension suggestions:

- As mentioned in the section on difference calculation, you may also want to experiment with alternate formulas for calculating the difference in color between two pixels. The edges will be sharper if you select a difference calculation where *difference(a, b)* does not equal *difference(b, a)* (for nonzero differences; i.e., produce a signed output), because then only one of the two pixels will decide it meets threshold to be an edge. One reasonable idea for a distance calculation would be Euclidean distance between RGB values treated as vectors of dimension 3.
- There are more advanced edge detection algorithms that you could use, especially if you are experienced in linear algebra. Wikipedia page for edge detection is a good place to start. Hough Transform is a really cool one that specializes in finding straight lines or circles (google it).

Green screen extension suggestions:

- Let the user specify an RGB value to use instead of green (e.g., sometimes you’ll see blue screen instead of green screen in filmmaking). This could be done by typing the RGB value, or, if you initially show the entire sticker in the window, allowing the user to click a pixel to select the color.
- Ask the user specify a resizing factor to scale the sticker image up or down for a better fit to the scene.
- Allow the user to flip the main or sticker image or horizontal/vertical.

Scatter extension suggestions:

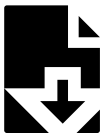
- There are many more interesting blur/smudge type algorithms, such as incorporating an average of neighboring pixels (for some radius of neighbor). Research and implement one of these algorithms. Let the user specify one specific area to scatter, rather than the whole image. This could be useful for anonymizing images by blurring a faces or private data.

General extension suggestions:

- Create composite filters that automate multi-step operations (e.g., first do edge detection on an image that you will then use as a sticker in a particular place, etc.). You could even insert pauses between the executions of each step, so it looks like a simple animation. To add a pause, use the function **`pause(miliseconds);`**.
- Add various color effects filters: grayscale, b&w, sepia tone, negative. All of these are very simple manipulations of the RGB values for each pixel.
- Allow users to resize the image. Unlike the others it does not preserve each pixel exactly, so you'll need to think through the algorithm for that (there are lots of nice and easy options—google to research).
- A really neat trick that is surprisingly easy is steganography combination of two images. You'll want to include an option to both encode a message/image, and recover the secretly encoded message/image.
- Add a GUI menu interface (must be in addition to our console-based menu interface).
- Add brush/pen/pencil drawing on the photo using the mouse.
- Add bucket fill, or bucket fill with a tolerance for non-exact colors (using your color difference calculation again). The algorithm to fill an area is something we'll cover later in the quarter, but you could read ahead if you are interested in trying this.

# Starter Code

Click the icon below to download the starter code:



Starter Code

# Sample Output

Click here for sample output files that you can check against your own output.

# Deliverables

Turn in the following files:

1. Code: **`fauxtoshop.cpp`**, the C++ code for all of your filters.
2. Art: **`art1.jpg, art2.jpg, art3.jpg`**. Three images that you created using your code (use the save resulting image option to save the file). You should use several of the filter effects in combination (e.g., multiple green screen stickers, one of them edges only or scattered), by saving interim files and reading them in as input to the next step. Although you may use some of the images provided in the starter code to generate these three pieces of artwork, they must each include some of your own images that were not part of the starter code (e.g., your own photos, things you sourced from the internet). You may use some of your extensions (if any) in making these three artwork files.
3. *Optional*: Any extension C++ files and any additional image files that would be needed for the grader can operate your extensions options (if any). Be sure your code includes comments and/or cout statements that make it clear to the grader how to operate your extensions. **Imporatant: Please name your extension C++ files "extensionCPP" (without the period)**. This will allow the grader to first grade your original files, and then grade your extensions.

---