FlashPhoto (Iteration #1) Assignment Handout

Dear class, welcome to this first iteration of our software design and development project! As we discussed on the first day of class, we'll be creating a digital painting and image-editing tool, similar to some you have likely used before (e.g., Adobe Photoshop, Gimp, MS Paint, various smartphone apps). It's a fun program, and we think you'll really enjoy the visual results that come from your work!

1. The Story Behind Our Project

Our "story" will be that you have just joined a software development team for a project called FlashPhoto. The other members of the dev team (represented by us, your CSci-3081 instructors and TAs) have been working on the project since its conception a few months ago, and so far we have built an initial digital painting application that you will be able to clone from our shared-upstream repo to use as your starting point. Together, we now have three weeks to complete the next phase of the project, which focuses on adding several new features to the tool: (1) image filters like blur, sharpen, edge detection; (2) support for undo and redo; and (3) a "blur" tool to complement the simpler existing tools like "pen", "chalk", and "spray can". Since we know today is the start of your very first week on the project, we expect that your focus for most of this week will be learning the existing code base. So, as part of this, we have asked you to create a good UML class diagram of the existing code for us that we can add to our project documentation – we've been meaning to do this for a while, and it's a great way to learn the code. After this, we're planning to have you jump right in to implementing the image filters. At the same time, we will be working on the undo/redo and blur tool features. We'll use git to manage our code so that we can all work simultaneously and merge our work together as we progress.

2. Background on the Existing Code Base

Since your first job is to understand the existing code base, we'll present some background here on the existing application, which includes 6 simple digital painting tools (pen, chalk, calligraphy pen, highlighter, spray can, and eraser) that can be operated with the mouse. The current tool and other parameters, such as the tool color and size,

are set using a control panel that floats inside the main application window on top of the digital "canvas" where users draw and paint using the tools.

What design decisions have been made so far? It is useful to understand a few of the requirements that have driven the design we have created thus far. Here are some of the key ones:

- The code requires that there is always one (and only one) currently active `Tool`.
- Each specific `Tool` is implemented in its own subclass (e.g., `ToolPen, ToolChalk, ToolEraser`).
- When the user clicks and drags the mouse on the digital canvas, the current tool "applies itself" to the canvas. For most of the tools this adds paint to the canvas, changing the red, green, and blue values for the underlying image pixels, which are stored in a `PixelBuffer` object. However, some tools, such as the eraser, have special behavior.
- All tools are implemented using a `Mask` to represent the shape and intensity of the tool when pressed against the canvas. This shape is stored as a 2D array of floats using the `FloatMatrix` class. Some tools are shaped like a circle and others are rectangular. Some have a constant intensity and others vary spatially; for example, the intensity of the paint from the spray can is high in the center but falls off away from the center.
- For performance reasons (updating lots of pixels multiple times a second can be slow), the Mask used for each paint stroke made by the user is pre-calculated once immediately after the user presses down on the mouse button. Then, this mask is used repeatedly to continue adding paint to the canvas as the mouse is dragged around the screen.

What do each of the tools do? Here are some more details on the specific tools. You should also try them out for yourself to learn more!

Pen: This tool is designed to look like a pen. It makes an opaque, circular mark with constant intensity. This means that when the pen is applied to the canvas the digital ink from the pen completely covers any color that had previously been applied to the canvas at that position. The color of the pen can be set interactively using the GUI.

Calligraphy Pen: This tool is designed to look like a calligraphy pen. It works almost the same way as the Pen tool. Like the Pen tool, it is completely opaque and it covers up any "paint" that was already applied to the canvas. However, the difference with the Calligraphy Pen is that the mask is an oval tilted at a 30-degree angle. The color of the highlighter can be set interactively using the GUI.

Chalk: This tool is designed to look like a piece of chalk dragged across a bumpy surface. It has a circular mask, but about 40% of the pixels within the circle are randomly chosen to be completely transparent. This creates the bumpy effect. The chalk color can be set interactively using the GUI.

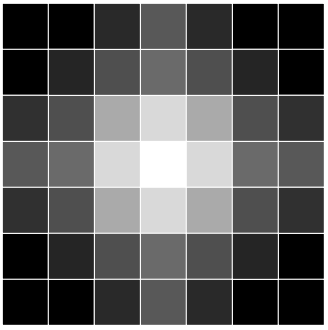
Highlighter: This tool is designed to look like a highlighter marker. It uses an oval mask similar to the Calligraphy Pen, but the oval is oriented vertically, and the intensity of the mask is set to make the mark semi-transparent. The color applied to the canvas is approximately 40% the color of the highlighter and 60% whatever color is already on the canvas; however, a special color blending function is used for the highlighter so that dark colors show through more than light colors. The color of the highlighter's ink can be set interactively using the GUI.

Spray Can: This tool is designed to mimic the look of paint from a spray can. It uses a special mask that is circular in shape but has a linear falloff to mimic the dispersion of color from the spray. The intensity is strongest at the center pixel and falls off linearly to zero intensity at the edge of the circle. The specific paint color can be set interactively using the GUI.

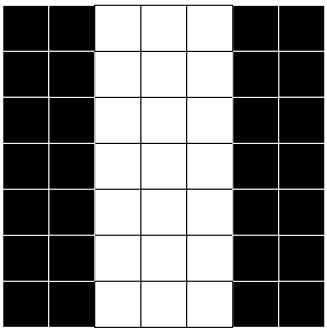
Eraser: This tool erases the digital canvas and returns it to its original background color. It has a circular mask, and eraser's color cannot be changed through the GUI because there is currently no user interface for changing the canvas background color. (Feel free to add one if you wish.)

What exactly is a Mask and how does it work? Each mask holds a 2D array of floats that, similar to the way a stamp works, determine the shape of the tool when pressed against the canvas. In addition the shape, masks also encode the intensity of the color to apply. For some tools this is constant, but for others, like the spray can, the intensity changes as a function of the distance from the center of the mask. Our implementation assumes the masks are square and that the width and height are odd numbers. This makes it easy to identify the center point of the mask.

The images below show two examples masks, each 7 pixels wide x 7 pixels high. In this figure, we've coded each float in the 2D array using a gray scale value, where 0.0=black and 1.0=white. In the example on the left, notice how the colors get darker as they move away from the center, so this mask demonstrates something similar to the linear falloff used for the spray can tool described earlier. In the example of the right, notice that the each element is the mask is either 1.0 or 0.0. This mask encodes a rectangle that is 3 pixels wide by 7 pixels high, a simplified version of what is used for the Highlighter and Calligraphy Pen tools.



Mask encoding linear falloff.



Mask encoding a 3x7 rectangle.

Think of the mask as a small invisible window that is centered at the position of the mouse cursor and slides over the top of the digital canvas with the mouse as the mouse moves. The center pixel for the mask always lies directly under the mouse cursor, and each pixel of the mask will line up perfectly with a corresponding pixel of the underlying canvas, but the correspondence between mask pixels and canvas pixels will change as the mouse moves.

The value that the mask stores at each element in its array is a floating point number between 0.0 and 1.0. This indicates the "intensity" that the tool should have (i.e., the amount of color it should apply) on the canvas at that pixel.

Why bother with a mask? The answer is speed. Calculating the distance between two points involves several mathematical operations (some additions, some multiplies, and a square root). We don't want to have to repeat this calculation for every pixel under the tool every time the tool is applied, which should be at a speed of about 30-60 times per second in order for the painting to feel smooth – that would be a lot of computation that could slow down our program, and once the tool's radius is set, these values never change, so we really only need to compute them once each time the radius is set, or, as we do in the current implementation, once at the start of each stroke.

Do all tools use their mask in the same way? This is an excellent question. Clearly, not every tool will use the same mask, but our design enforces that every tool "has a" mask, and the way that the mask is applied to the canvas is almost exactly the same in all cases. The two or three exceptions to this rule make the program design a challenge, and we'll discuss these at length in class, debating pros and cons of various design approaches.

3. New Feature: Eight Image Filters

You will be responsible for adding image filters to FlashPhoto. This is a very exciting addition to the program. With image filters, we'll now be able to blur, sharpen, or adjust saturation levels in photographs or even automatically detect all the edge features in an arbitrary image.

Original Image

Image filtering is a complex topic, but it turns out that many impressive filters can be created quite simply. Filters 1-4 are the easiest. These just adjust the color of each pixel in an original image using different mathematical equations, which we'll describe in detail in class. Filters 5-8 are convolution based. We will also discuss this approach in class, so please refer to the slides from class for more technical details. Convolution filters are extremely powerful and really fun to implement!

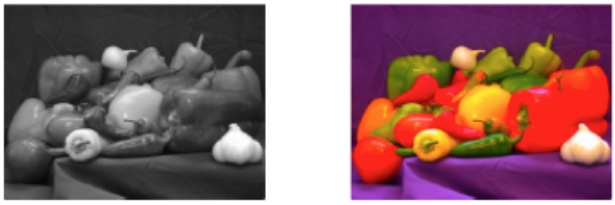
1. Threshold Filter

Each of the color channels will be rounded up to the maximum value of 1.0 or down to the minimum value of 0.0 based on whether the pixel's value is greater or less than the GLUI input value, respectively.

Thresh(0.1) Thresh(0.5) Thresh(0.7)

2. Saturation Filter

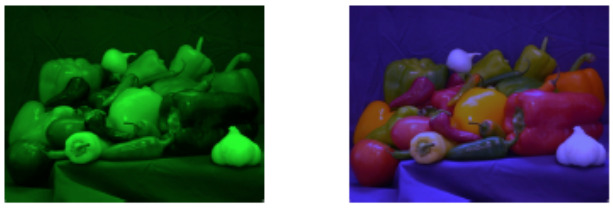
Increase or decrease the colorfulness of the image.



Saturate(0) Saturate(2)

3. Channels Filter

Independently scale up or down the intensity of each color channel (red, green, and blue).



Channels(0,1,0) Channels(0.5,0.5,1.5)

4. Quantize Filter

Reduce the number of colors in the image by binning. If using 4 bins, there will only be 4 possible intensity values for each color channel, spaced evenly: 0%, 33%, 66%, and 100%. Adjust each color channel value R, G, and B to put it into

the nearest bin.


Quantize(2)


Quantize(4)


Quantize(8)

5. Blur Filter

Use a Gaussian Blur image convolution kernel to blur the image in proportion to the amount specified in the GUI.


Blur(5)


Blur(10)


Blur(20)

6. Motion Blur Filter

Blur the image by convolving it with an appropriate motion-blurring kernel. Support four possible blurring directions (North-to-South, East-to-West, Northeast-to-Southwest, and Northwest-to-Southeast) and blur according to the amount specified in the GUI.


MotionBlur(5,N/S)


MotionBlur(10,NE/SW)


MotionBlur(20,E/W)

7. Sharpen Filter

Sharpen the image (accentuate the edges of shapes) in proportion to the amount specified in the GUI by convolving the image with an appropriate sharpening kernel.


Sharpen(5)

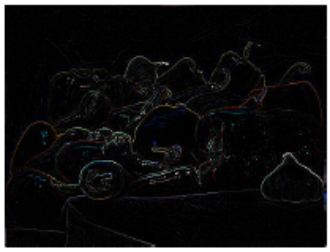

Sharpen(10)


Sharpen(20)

8. Edge Detection Filter

Transform the image into one that only shows the edges of shapes by convolving

it with a 3x3 edge detection kernel. Pixels on the border of differently colored regions will be bright, while pixels in areas of low change will be dark.



Edge Detect

4. Prelimary and Final Handins

To help you stay on track with the longer-term assignments you'll have in this course, we will break each assignment into multiple preliminary deadlines and one final deadline. You will be graded on the materials in your github repo master branch only at the time of the final deadline, but we strongly suggest that you work on the assignment a bit each week, pacing yourself according to the suggested timeline. At each preliminary deadline we will add some tests to the automated gitbot feedback scripts to give you a sense of how your current implementation matches the expectations we'll use for grading. At the final deadline, the feedback tests will be used are part of our automated grading, but we will also add some new tests that you do not get to see. The automated grading will make up just one portion of your final assignment grade. You'll also be graded on your writing, include design documents, diagrams, and code organization and style. More info on assessment is provided at the end of this document.

4.1 Iteration 1, Preliminary Handin #1 (after ~1 week)

- **Merge the support code into your repo and check feedback**

The first step you should take to start the project is to git pull from shared-upstream to merge the support code into your repo. Once you do this and commit and push the changes to your github origin, you should see that a new feedback file has been generated. The code should pass all of the feedback tests.

- **Complete support code UML diagram**

Following our "story", we suggest a focus in week 1 on understanding the code by reading through the entire project code and creating a UML class diagram for the entire project. This UML diagram will be part of your final handin.

The requirements for the UML diagram are:

1. Construct a single UML diagram that captures the design of, and relationships between, each of the classes in the source code.
2. Use a level of detail in the diagram consistent with Figures 3.1, 3.2, and 3.3 of Fowler's book.

    - Use proper UML for the name, attributes, and operations class sections.
    - For associations and generalization, show correct directionality, name roles, and multiplicities, and distinguish between containment-style associations ("has a") and generalization-style ("is a") by using a regular arrowhead versus an open triangle arrowhead. However, there is no need to go to the level of specifying whether the association is a form of aggregation or the other nuances that can be indicated via additional different types of arrows.
    - Make use of italics to signify any virtual methods and abstract classes.
    - Make use of notes when they would help a reader better understand or remember what is going on in the code (which is, after all, one of the purposes of the exercise).
    - You do *not* need to include public/protected/private indicators (although you are welcome to do so if you wish).
    - If you have a variable listed under attributes and the class also contains methods for get_variable() and set_variable(), then you don't need to list the variable as an attribute and the accessor and mutator as operations. List the variable, or list the accessor and mutator — choose whichever option is more informative.
    - You do not need to diagram MinGfx::GraphicsApp() or any other classes that are not defined directly in the support code files we provide.

3. You are encouraged to draw your diagram by hand (physical sketching helps you learn), but you can also use Powerpoint, Keynote, Visio, Gliffy, dia, Google Diagrams, or any other diagramming software if you prefer. In either case, ensure the final version of the diagram is clean, clear, and concise.

4.2 Iteration 1, Preliminary Handin #2 (after ~2 weeks)

- **Merge in expected code from your teammates (us), resolve any conflicts**

Watch for updates to the shared-upstream repo this week. Remember we, the other members of your dev team, will be working on the undo/redo features and a special new tool called the "blur tool". We'll push our code for these features during the week, and you should pull to integrate them into your own repo. We wouldn't expect any major merge conflicts, but it's always a possibility when you're working as a team. If they come up, resolve them as you practiced in homework earlier in the semester.

- **Start on implementing the image filters!**

Start with filters 1-4 described in Section 3 of this document; these are a bit easier. After this, move on to the convolution filters.

4.3 Iteration 1, Final Handin (after ~3 weeks, see Canvas for deadline)

- **Finish implementing all eight filters**

You should have a good start on filters already by the start of this week. Finish them up and test them out on some images!

- **Finish the Blur Tool by connecting it to your blur filter**

Remember, we're in charge of implementing the blur tool class, but we can't totally complete this task until you have finished implementing your blur filter, since the tool uses the filter by applying it to just a small region of the image under the mouse. After you finish your blur filter, remember to follow the comments we leave for you in the `ToolBlur` class to finish the implementation of that tool.

- **Write some tests using Google Test**

Following the guidelines and examples you studied in the Google Test homework assignment earlier in the semester, add a minimum of three unit tests to the project. You'll have to much more thoroughly test all your code in future iterations, so if you want to get a head start on that, please feel free to add more tests. However, since we're just getting started, in this iteration we'll grade not on the quality of your tests but just on the presence of 3 or more unit tests – at this stage, we just want to make sure you know how, technically, to add a test to the project.

- **Remember to hand in everything else too!**

Don't forget, the final handin is when we will actually grade everything, even the preliminary handins. So, make sure your repo is complete with all of the bolded bullet items listed in sections 4.1 to 4.3.

5. Grading

You can use this high-level grading rubric to better understand how we will grade the project and as a checklist to make sure you have completed everything.

50% - Program Design (Typically graded by hand by inspecting the code)

- UML Diagram
    - The diagram shows all the classes.
    - The relationships between the classes are correctly shown and include multiplicities when appropriate.
    - The class attributes, operations, etc. are present and correctly shown.
    - The diagram includes notes to explain the code in at least one or two situations where you feel this would help readers understand the organization of the code.
    - The diagram is legible to an audience that understands UML; classes are arranged on the page without creating a bird's nest of crossing lines.
    - Generally, UML is used correctly and informatively.
- Code follows the well-designed solution discussed and diagrammed in class.
    - We will discuss pros and cons of various approaches to implementing the filters in class, and, together, we'll come to an "instructor approved" solid design, which we will convey clearly to you using UML diagrams or similar mechanisms. You're solution should implement this design rather than coming up with your own approach.
- Code follows good design and style practices as discussed in class.
    - Class interfaces follow McConnell's guidelines of implementing just a single abstract data type within each class and using consistent levels of abstraction.
    - Class and important variable names are informative.
    - Objects that are dynamically allocated using new are later deleted and pointers are set to NULL when they do not point to valid memory locations.
    - Comments are included to describe the intent of each public member function in every class.
    - Additional comments are included in areas where the intent of the code is not obvious from reading the code itself and where future programmers would encounter a special case or unusual aspect of the code that they would find surprising.

50% - Program Functionality and Robustness (Typically graded by running the code, either using scripts or interactively)

- The program builds without error by running "make" in the root directory of the project and runs on the CSE Lab machines.
- Code for the undo/redo feature is correctly merged into your program.
- Four simple filters are implemented according to the specs described earlier and equations presented in slides from class.

- Four convolution are filters implemented according to the specs described earlier and equations presented in slides from class.
- Implementation of the Blur Tool is completed by merging our ToolBlur code and connecting your blur filter to it.
- The project includes at least 3 Google Test unit tests that you have written yourself.
- The project follows Google Style as demonstrated by passing the cpplint.py checks.