FlashPhoto, Mia, & ImageTools (Iteration #2) Assignment Handout

1. Introduction

Congratulations on completing Iteration #1 of the semester project for CSci-3081W; now on to Iteration #2!

Surprise! As you found out in class, it turns out we're no longer a company that makes computer tools for artists; our management has decided that now we are supposed to make medical imaging software while also maintaining our original FlashPhoto tool for artists! This type of drastic shift in direction happens all the time in software construction. With this addition to "our story" in mind, let's look at what our team needs to accomplish in Iteration #3.

2. Requirements

There are four programming-oriented tasks in this iteration. These are described in the next sections.

2.1 Refactor to Create a Library plus Two Applications

The first main task given the new direction we are taking is to refactor the code. Since we now have two different applications to support, we'll want to build each of these applications in a separate directory. However, note that both applications will depend upon some common classes and infrastructure. It would be difficult to maintain (and generally bad programming practice) to replicate this code within the two applications, so instead we will create our own library (`libimagetools.a`) to hold all this common code. Then, we can have the two applications we build refer to our own `libimagetools`.

Here are some specific requirements:

- Refactor and reorganize your code to create a new library (`libimagetools.a`) and two applications (`flashphoto` and `mia`) that each build in a separate directory.

- Use the guideline that you shouldn't have any duplicated classes or code. In other words, anything that is common to both `flashphoto` and `mia` should be placed in `libimagetools` so that it can easily be used by both applications.
- When `make all` is run in the imagetools directory, a library called `libimagetools.a` should be built. To do this, your Makefile will need to be changed to create a *library* not a *program*.
- The `flashphoto` application should essentially be the same application that you handed in for iteration #1, but it will be compiled a bit differently. It should be placed in a directory called `flashphoto` and this *program* should be compiled when you run `make all` inside that directory. The `flashphoto` directory will therefore contain a `Makefile`, a `main.cc`, and maybe a couple of .cc and .h files for `flashphoto`-specific classes.
- Following a similar structure, we will create a new `mia` application in its own directory called `mia`. (The TA/instructor members of your team will be in charge of creating the GUI for `mia`. Watch for updates to shared-upstream that you can merge into your code.
- For convenience, you should include a project-level `Makefile` in the `PROJ` directory. Running `make all` from the root directory of your project should go into each subdirectory (in the order required, e.g., `libimagetools.a` needs to be compiled before `flashphoto` and `mia`) and run make there so that the whole project can be compiled by scratch by running a single command.

Here is a summary of the directory structure that we envision when this refactoring is complete:

```
Makefile
doc/
resources/
src/imagetools/
     Makefile
src/flashphoto/
     Makefile
src/mia/
     Makefile
src/tests/
     Makefile
```

2.2 Add support for Loading/Saving images

We can't go much farther in this project without an ability to load and save images, so we better add that functionality in this iteration. We (the other members of your team) will take the lead on implementing this functionality. We've written a simple image input / output library that you can link with your code.

2.3 Create a New Mia Application

The `mia` application is powerful but much smaller than `flashphoto` since it is a very special-purpose tool. The thing that makes this application so interesting is that it can run in two different modes. If you start it without specifying any command-line option, `mia` should run in a graphical mode that looks like a streamlined version of `flashphoto`. When a command-line argument is specified, `mia` should run in a command-line mode where this is no graphical user interface. For example, running `mia in.png -sharpen 5 out.png` will produce a new image out.png that is a sharpened version of `in.png`. This mode is useful for scripting and creating workflows that can be applied repeatedly to large medical imaging collections.

The two modes of `mia` should support the following features:

Graphical Mode:

[We will help with this; you'll just need to merge in our code.]

- A stamp tool that is hardcoded to stamp a bullseye pattern, one circle inside another, that clinicians can use to mark important areas on the image.

[These, you should already have working from iteration 1.]

- A pen tool, also useful for annotating images (e.g., circling a break in a bone).
- A subset of the filters you have implemented previously that would be useful for medical imaging (blur, sharpen, edge detect, threshold, quantize).

[And, this is the new file input output functionality that we will help you implement by providing a library to link with.]

- File Open/Save controls.

Command Line Mode:

[This mode is all new, this will be the major "new programming" task in this iteration.]

- Display a help message for a –h argument or any set of invalid arguments. You can copy our example (below) or determine your own good format for a useful help message.
  - `mia –h`
  - `mia <any invalid arguments>`

- Load an image specified by the first filename and save it out to the second filename. With no other arguments specified, this is equivalent to copying the image file.
  - `mia in.png out.png`

- Run an image processing command before saving out the new file. For example, to sharpen an image using a kernel of radius 5 and save the output to a new image:
  - `mia in.png –sharpen 5 out.png`

- Run multiple image processing commands in order. For example, sharpen an image by amount 5, then run an edge detection filter, then save the output to a new image. (This example just shows two filter operations, but you need to support an arbitrary number of filter operations that are specified in order they should be applied.):
    - `mia in.png –sharpen 5 –edgedetect out.png`

- Here is the complete list of command-line arguments to support as reported by a sample help message, which you are welcome to copy and use as your own:

```
usage: mia infile.png [image processing commands ...] outfile.png

infile.png:          input image file in png format
outfile.png:         filename to use for saving the result

image processing comamnds:
-blur r:             apply a gaussian blur of radius r
-edgedetect:         apply an edge detection filter
-sharpen r:          apply a sharpening filter of radius r
-red s:              scale the red channel by factor s
-green s:            scale the green channel by factor s
-blue s:             scale the blue channel by factor s
-quantize n:         quantize each color channel into n bins
-saturate s:         saturate colors by scale factor s
-threshold c:        threshold using cutoff value c
-motionblur-n-s r:   north-south motion blur with kernel radius r
-motionblur-e-w r:   east-west motion blur with kernel radius r
-motionblur-ne-sw r: ne-sw motion blur with kernel radius r
-motionblur-nw-se r: nw-se motion blur with kernel radius r
```

2.4 Testing

In iteration 1 you wrote a couple of very simple tests. This wasn't enough to seriously test the code but just enough to verify that you know how to write tests. Now, it is time to practice some more serious, organized testing. The specific testing requirements are:

Overall structure:

- Implement your tests using Google Test and by adding files to the `tests` directory, as we have been practicing so far.
- Running "make all" inside the `tests` directory should run **all** the tests.

Unit tests:

- Write a series of unit tests to thoroughly test your command line parsing.
- Test to make sure that you command line parsing routines generate the expected command list for valid command lines.
- Test to make sure you catch errors in the command line so that you can respond by printing out a help message.

- At a minimum, you should write enough tests to cover:
    - Testing to make sure you detect an invalid command line for common errors or typos that users might make, such as forgetting to include the input or output file or leaving off an argument for commands that require arguments.
    - Testing to make sure you calculate the correct set of image processing commands to run for valid command lines that list 0, 1, or multiple image processing commands.

Regression tests:

- For regression tests, the goal is to verify that features that worked once stay working, even as you continue to add to the code.
- For a library that involves manipulating images, this means that once we have the blur filter working, we should expect that if we run blur with radius=5.0 on Monday on an image, `test-input.png`, and save the result to `blur-5.0-monday.png` and then on Wednesday we run blur with radius=5.0 on the same `test-input.png` and save the result to `blur-5.0-wednesday.png`, we should see that the two resulting images are exactly the same (i.e., each pixel in `blur-5.0-monday.png` is exactly the same color as the corresponding pixel in `blur-5.0-wednesday.png`).
- To use this testing strategy with Google Test, you should add a `PROJ/resources` directory to your repo where you can store images. Then add some `PROJ/resources/test-input.png` that you can use as a consistent test input image. Then, for each feature you wish to test, you need to create and save an expected output image to treat as the "ground truth". You can come up with your own naming scheme for these, for example, you might store the ground truth for a blur filter with radius=5 as: `PROJ/resources/blur-5-expected.png` or `blur-5-output.png` or `blur-5-truth.png`. Finally, after this initialization, you can go ahead and write your tests. Each test should load `PROJ/resources/test-input.png` into a pixel buffer, apply the feature you want to test, then load the corresponding expected output (e.g., `PROJ/resources/blur-5-expected.png`) into a second pixel buffer and compare the two pixel buffers (note that `PixelBuffer` has an overloaded `==` operator) to see if they are equal.
- You should test all of your filters and all of your tools using this strategy. For the tools, note that you can "fake" user input from the mouse by calling `StartStroke()`, `AddToStroke()`, and `EndStroke()` directly from your testing code.

3. 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, including design documents, diagrams, and code organization and style. More info on assessment is provided at the end of this document.

3.1 Iteration 2, Preliminary Handin #1 (after ~1 week)

- **Revise project directory structure and makefiles.**
  You'll need to `git mv` all files except for `flash_photo_app.[h,cc]` and `main.cc` over to a new `imagetools` directory. Also, remember to adjust your makefiles and add new ones as described earlier.

- **Refactor, following a MVC paradigm, to pull key functionality out of the FlashPhotoApp class and put it in a new ImageEditor class that can act as the underlying "Model" for the FlashPhoto and Mia view-controller pairs.**

  We will help with this step by pushing the header file for a new `ImageEditor` class to shared-upstream. You should be able to create a corresponding `.cc` file almost entirely by copying and pasting code that you originally wrote for the `FlashPhotoApp` class. Since it is not good to replicate the code in multiple places, you'll also need to update `FlashPhoto` to call `ImageEditor` for all of the functionality it needs to implement. This will make for some well-designed code where the GUI is loosely coupled to the rest of the program, having no responsibilities other than calling functionality in the `ImageEditor` class, which can also be used by other views or controllers, such as `mia`'s command line.

3.2 Iteration 2, Preliminary Handin #2 (after ~2 weeks)

- **Link with our imageio library to add image saving/loading to both programs.**

  The imageio library is installed on the CSELabs machines as follows:

Library name:	libimageio.a
Library directory:	/classes/csel-f18c3081/lib
Include files directory:	/classes/csel-f18c3081/include/imageio
Documentation:	/classes/csel-f18c3081/doc/imageio/index.html

  Edit your makefiles to link it into your programs so that loading and saving functionality can be included in both FlashPhoto and Mia. Since the GUI design has been one of our primary roles, we'll add some buttons to FlashPhotoApp that you can

use to call this functionality, and you'll see almost the same thing will be in the Mia app (described next) as well. Check out the imageio documentation for info on how exactly to use the library, and use this as a chance to practice the important skill of linking a 3rd party library into a project.

- **Hook up the graphical mode for the mia app.**

  We'll take care of implementing the mia graphical user interface, getting all the buttons in the right place, etc., and pushing this code to shared-upstream, but you will need to hook the GUI up to the functionality in your ImageEditor in order to make it work.

3.3 Iteration 2, Final Handin (after ~3 weeks)

- **Implement the command line mode for the mia app.**

  We'll contribute one final aspect to the code to help with this task. Following the Command Pattern, we'll push a set of classes to implement ImageEditorCommands that you can use to help you implement command line parsing for mia. You'll still need to do all of the command line parsing yourself, but once you determine the correct command(s) to run based on the command line arguments, you will be able to use our collection of ImageEditorCommnds to store a list of commands to run.

- **Testing.**

  You need to get serious about testing in this iteration. Implement both unit and regression tests as described earlier in the document to test the command line parsing, filters, and tools.

4. 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)

- Code follows the well-designed solution discussed and diagrammed in class.
  - We will discuss pros and cons of various approaches to refactoring in class, and, together, we'll come to an "instructor approved" solid design. You're solution should implement this design rather than coming up with your own approach. For this iteration, the most important aspect will be following a MVC paradigm during refactoring.

- 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.
    - The tests show evidence of thinking about aspects of the design that are "worth testing" (e.g., boundary conditions, areas of change).

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

- The library and both programs build without error by running "make" in the root directory of the project, and the two programs run on the CSE Lab machines.
- The code links with the imageio library to handle image loading and saving.
- The mia mode graphical is functional and calls routines in the imagetools library to implement the image editing operations.
- The mia command line mode is functional, responding to a variety of valid and invalid arguments.
- Both unit tests and regression tests are implemented.
- Tests cover valid/invalid command line options.
- Tests cover different numbers of options specified on the command line.
- Tests cover all tools and filters.
- The project follows Google Style as demonstrated by passing the cpplint.py checks.