Project Release (Iteration #3) Requirements and Group Handin Procedures

1. Introduction

Congratulations on completing the first two iterations of the semester project for CSci-3081W; now on to our final iteration, which we call our Project Release. You will be working on a number of new technical features during this iteration, but as implied by the name, the emphasis in this iteration will be on getting your project to the point where it is such a solid platform and code base that it is ready to be used by both end users of the software and other developers outside of your team. Please note that part of reaching this stage of software refinement is writing – documentation, guidelines, tutorials, etc. You've practiced this on your own all semester through our weekly writing assignments. Now, you'll be writing as a group, in a much more "real world" context. As you plan your schedule with your group for this iteration, make sure you keep the importance of this writing in mind. For this iteration, the writing aspects of the project are just as important as the technical aspects, and this will be reflected in the grading.

Now on to the requirements!

2. Programming and Writing Requirements

Surprise! We hope you enjoyed the dramatic reveal in class as we introduced our final iteration of the project. (If you missed it, 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 keeping our old product line alive!) This type of drastic shift in direction happens all the time in software construction; it's not unusual. So, you need to plan for it. If you have done a good job at designing your software for the first two iterations, then this change shouldn't actually be that difficult to accommodate. If, on the other hand, you've just hacked together some code that works, but doesn't have a good design... then, you may have some trouble in this iteration ☺. (Remember, you can always use our solution to Iteration #2 as your starting point for

Iteration #3, so this shouldn't really be a problem for the class – but, if you find yourself in this situation, please do make sure you learn from the experience!)

With this motivation in mind, let's look at what you need to accomplish in Iteration #3.

2.1 Programming and Code Refactoring

There are three main thrusts to the programming-oriented tasks in this iteration. These are described next.

2.1.1 Programming Thrust 1: Create a Library plus Two Applications

The first main task given the new direction we are taking is to refactor your 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 (bad programming practice) to replicate this code within the two applications, so instead we will create our own library (libphoto.a) to hold all this common code. Then, just the same way that Photoshop refers to libjpg or libpng, we can have the two applications we build refer to our own libphoto.

Here are some specific requirements:

- Refactor and reorganize your code to create a new library (libphoto.a) and two applications (PhotoShop and MedImage) 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 PhotoShop and MedImage should be placed in libphoto so that it can easily be used by both applications.
- When "make" is run in the libphoto directory, a library called libphoto.a should be built. Note, to do this, your makefile will need to be changed to create a *library* not a *program*.
- The PhotoShop application should essentially be the same application that you handed in for iteration #2, but it will be compiled a bit differently. It should be placed in a directory called PhotoShop and this *program* should be compiled when you run "make all" inside that directory. The photoshop directory will therefore contain a Makefile, a main.cpp, and maybe a couple of .cpp and .h files for photoshop-specific classes.
- Following a similar structure, you should create a new MedImage application in its own directory called MedImage. (The requirements for this new application are discussed in the next section.)
- For convenience, you should include a project-level Makefile in the root directory for your project. Running "make all" from the root directory of your project should go into each subdirectory (in the order required, e.g., libphoto.a needs to

be compiled before PhotoShop and MedImage) 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. This does not cover all the files and directories that will be present, but instead suggests the general structure.

```
makefile
web/
documentation/
libphoto/
     include/
     lib/
     makefile
glui/
libpng/
libjpeg/
PhotoShop/
     makefile
     photoshop
MedImage/
     makefile
     medimage
tests/
     makefile
```

2.1.2 Programming Thrust 2: Create a New MedImage Application

The MedImage application is powerful but much smaller than PhotoShop 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, MedImage should run in a graphical mode that looks like a streamlined version of PhotoShop. When a command-line argument is specified, MedImage should run in a command-line mode where there is no graphical user interface. For example, running "`medimage 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.

For the graphical mode, just as we did for iteration #2, the TAs will again post code to Moodle that will implement a GLUI interface that is appropriate for the project. Please download this code and incorporate it into your MedImage application. Using this graphical interface, your MedImage application should support the following.

MedImage Graphical Mode:

- A rubber stamp tool that is hardcoded to stamp a red marker for annotating medical images.
- A red 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 (sharpen, edge detection, threshold, quantize, convert to grayscale (same a saturate to 0), multiply RGB (same as channels))
- File Open/Save controls similar to the previous iteration, but with a small addition:
    - For 3D medical imaging data (e.g., a CT or MRI scan), the data often come as a "stack" of images (e.g., image001.png, image002.png, image003.png), so we want to make it easy to load these stacks.
    - To do this, the GUI will now support loading a directory of images.
    - In addition, the GUI now includes two buttons, Prev and Next, that will automatically load the previous image in the stack and the next image in the stack when clicked.
    - Changes to each image are only saved if "Save Image" is pressed. Changes to images are lost if the user switches to different image without saving
    - Sequenced Image names are always in the format of a name, a three-digit number, and an extension. Example: slice001.png, slice002.png, etc.
    - An image sequence ends when there is a break in the sequence names. Example: im001.png, im002.png, im003.png, im011.png, etc.
      If im001.png is loaded, the user can only press "next" twice.

Note that the GLUI interface distributed by the TAs will implement the new file loading functionality, and the rest of the features described above already exist in some form inside PhotoShop. So, this portion of the assignment is really about re-packaging rather than writing new functionality.

The new command line mode mentioned above is mostly re-packaging, but not entirely – you will need to handle reading and parsing command line arguments for the first time. This mode needs to support the following options, all specified on the command line when medimage is run from a command prompt in a shell.

MedImage Command Line Mode:

- DisplayHelp (determine your own good format for a useful help message):
    - `medimage –h`
- Convert an image in JPG format to PNG format (please also support png to jpg):
    - `medimage in.jpg out.png`

- Sharpen an image by amount 5 (corresponds to the parameter you can specify in the iteration #2 GUI) and save the output to a new image:
  - `medimage in.png –sharpen 5 out.png`
- 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):
  - `medimage in.png –sharpen 5 –edgedetect out.png`
- Apply commands to each image in a sequence:
  - `medimage in###.png –sharpen 5 out###.png`
    (This should apply the commands to every image with a name that matches the speficified pattern, and save each image with the same three-digits.)
- Finally, since it will be a useful utility for the automated testing discussed in the next section, also implement a very simple command that prints "1" if the two images are identical and prints "0" if they are different.
  - `medimage image1.png –testsame image2.png`
- Complete list of required command-line arguments:
  - `–h`
  - `-sharpen <integer>`
  - `-edgedetect`
  - `-thresh <float>`
  - `-quantize <int>`
  - `-blur <float>`
  - `-saturate <float>`
  - `-multrgb <float> <float> <float>`
  - `-stamp <int> <int>`
  - `-compare`
- Some helpful guidelines can be found here:
  http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html

2.1.3 Testing

In iterations #1 and #2 our use of testing was informal. Now, it is time to practice some more serious, organized testing. We will use a few different methods for testing our applications, including tests written directly in C++ and tests run via commands specified in a Makefile. The specific testing requirements are:

Overall structure:

- As much as possible, organize all of your testing-related code and scripts inside a "tests" directory.
- Running "make all" inside the tests directory should run **all** the tests.
- If a test fails, then an appropriate message should be printed.

- If a test succeeds, then it is ok to print nothing (for testing, no error report generally means success) or, if you prefer, you can print a success message.
- There are just 3 things you are responsible for testing:
    1. Test each of PhotoShop's interactive tools to make sure it produces an expected result when the brush is moved along a pre-defined path.
    2. Test each PhotoShop filter to make sure it produces the expected output given some known test input image.
    3. Test the command line options for medimage to make sure a variety of command line options are handled correctly, including bad user input.
- To gain experience with a variety of testing methods, you should use a mix of tests written in C++ and tests written directly into your Makefile to accomplish this – keep reading for specifics.
-

Testing via a Makefile:

- To test each libphoto filter (and learn a bit more about how to work with Makefiles), write at least one test for each libphoto filter using a separate target (e.g., "test_blurfilter", "test_edgedetection") within the Makefile.
- Remember, Makefile rules are sort of like scripts; you can run a series of commands as part of these.
- Here, the series of commands will need to call medimage (or another command-line application you've created) with appropriate arguments to run the filter on a test image, generate a result, compare the result with an expected output, and then report that the test succeeded or failed based upon this comparison.
- You will want to have a folder of expected images that you've verified as correct ahead of time.

Testing by writing tests in C++:

- To test each of PhotoShop's interactive tools, write at least one test for each interactive PhotoShop tool.
- Since these will be part of your whole "testing suite" they should also be called from your Makefile as part of what happens when you run "make all" in the tests directory, but to implement these tests, you'll need to use some C++ programming.
    - If you want, you can implement these tests using the cxxtest framework we discussed in class.
    - Or, you can write your own separate, small C++ program for each of these tests.
    - Or, you can write your own larger C++ program that includes all the tests.
- The testing code you write in C++ should generate "fake" user input, e.g., calling the tool's applyToCanvas method repeatedly according to some pattern of movement.

- This will produce a modified PixelBuffer.
- The test should then compare the modified PixelBuffer with an expect result, for example, a pre-saved image.
- While the "fake brush input" portion of the test must be written in C++, it is ok, if you prefer, to do this last portion of the test (comparing the result to a pre-saved expected result image) directly in the Makefile or script that runs the test. You'll have to think about the best way to design this testing strategy.

Testing by either C++ or a Makefile, whatever you think is most appropriate:

- The final thing to test is the command line parsing aspect of the software.
- Test valid inputs to make sure that medimage makes the right decision about which filters, etc. to run.
- Also test invalid inputs to make sure the error handling is reasonable. (When it encounters bad input, medimage should exit with an error (in C++ call exit(1)) and print the help message (same as running with a –h argument).
- Use whatever method you think is most appropriate to accomplish this testing.

2.2 Writing to Support a "Project Release"

We will do all the writing-oriented aspects of this iteration as webpages. Your assignment is to create a directory named "web" that is part of your project and make this the root for a project webpage that contains several things:

1. A main project page (index.html) that directs visitors to the different sections of the website, with a main distinction being resources for users vs. resources for developers.

2. For users: A set of three tutorials with links to example data.
    a. "Getting Started with PhotoShop" discusses brushes and filters.
    b. "Getting Started with MedImage" discusses loading images and annotating them.
    c. "Converting and Segmenting a CT Scan Using MedImage" discusses how to convert an image stack saved as a set of numbered images in a directory to a quantized version that has 4 tissue types. The approach for this should be to first slightly blur the image (to reduce noise), then quantize the result into 4 segments, then save the resulting images to a new directory. The tutorial should include a link to some sample data that can be downloaded in order for readers to follow through the exact steps described in the tutorial. (We will post a link to some sample data on Moodle.)
    d. Note: Remember to write these for users – that's a different audience than developers.

3. For developers: A link to documentation for all the code in the project (the library and the 2 applications) that you have auto-generated using doxygen. (Doxygen is a really cool tool that we will discuss in class that generates html documentation automatically from C++ code.) Our rule of thumb will be that all of the code in the project needs to be documented appropriately so that another programmer (someone not on your team) could use the code.

4. For developers: A webpage called "Design and Coding Style Guide" that contains sub-pages on:
    a. Project Design: A discussion of the overall design of the project, using UML-like diagrams to illustrate major classes so that new developers can quickly understand the structure of the project.
    b. Adding a New Brush: A tutorial on how to add a new brush to Photoshop.
    c. Adding a New Filter: The same but for a filter.
    d. Coding Style and Documentation: A discussion of the most important rules of coding style that you enforce within the project. This should include conventions for naming variables, documenting code, use of namespaces, etc.
    e. Programmers' Blog: Each member of your team should write his/her own blog entry that describes a section of code that you have personally written as part of the project. For each code snippet (this can be a single routine, perhaps 20-25 lines of code), copy and paste the actual code into the blog entry, describe its purpose, and how the code accomplishes this purpose, then describe the way that you decided to document the code. The idea here is that new programmers who join the project should be able to learn about appropriate styles for coding and documentation from these "great examples of existing code".

3. Group Handin Requirements

There are 2 handins to remember:

Sun 4/12, 11:55pm	Revision of Team Policies and Expectations, most importantly a schedule and plan for Iteration #3.
Sun 5/10, 11:55pm	Your Project Release, which includes: 1. The source code for your program. 2. The writing assignments (inside a directory called web). 3. Your Group Design Document (inside a directory called documentation).

(Reminder: No late work will be accepted. We will post our solution to the assignment online the day after it is due.)

Read the sections below carefully for instructions on each of the handins.

3.1 Team Policies and Expectations

This is the same document that you prepared with your team and turned in for Iterations 1 and 2. If everything worked well for you so far, then you might not even need to revise the policies and expectations. On the other hand, if you would like to make a change to help your group work more effectively, then this is the right time to do that.

Either way, one thing you **will need to update** is your group's plan/schedule for Iteration #3. Create this in the same style as before, including intermediate milestones and treat it as an attachment to your Policies and Expectations document. Handin the Policies and Expectations along with the attached plan/schedule on Moodle by the end of the first week of work on Iteration #3, that's by Sunday 4/12 at 11:55pm.

3.2 Your "Project Release"

The rest of your group's assignment is due on Sunday 5/10. Make sure that your handin contains all three parts of this assignment: 1. The actual source code that you have written, 2. The related writing assignments, 3. The design document.

You can turn all of this in in a single .tar file, which should be uploaded to the link provided on Moodle.

3.2.1 Source Code

Inside your .tar file, the source code can be arranged as you see fit, as long as it meets the requirements specified above (e.g., the source code for the library should be in its own directory, and the code for the two applications should be in separate directories).

Remember: You must ensure that your code will successfully build and run on the CSE Labs UNIX computers when you submit your source code. This will be vital for grading.

3.2.2 Writing Assignments

Inside your .tar file, your writing assignments should be placed inside a directory called "web" as specified above.

3.2.3 Group Design Document (PDF)

As in past iterations, inside your .tar file, your Group Design Document should be placed inside a directory called “documentation”.

For this iteration, you will be describing most of your important design decisions within the html pages you create. So, the Group Design Document will be brief and just needs to include Page 1 in the same style that you have created for the past iterations. For convenience, the instructions for Page 1 are repeated here:

Page 1 of the Group Design Document:

This is a cover page that should have the following 4 bits of information:

1. Names of group members.

2. Name of github group repository.

3. Statement of completeness of the solution. Ideally, this will just be a single sentence saying that you believe that you have completely and correctly implemented PhotoShop according to the specifications provided. If your implementation does not work entirely as intended, then please list what works and what does not work to help us understand how far you got in the project and give you as much credit as possible for the work that you completed.

4. A self-assessment of contributions to the group by each member. Here’s how this works: Your group gets 3 stars (*) for each member of your team. If you have a 3-person team, this means you have 9 stars in total. If you have a 4-person team, you have 12 stars in total. The stars belong collectively to the whole team. At the end of the iteration, you get to give these stars to individual team members as a reward. You have to give out all the stars, but you don’t have to give an equal number to each group member. Give as many as you think that group member earned given his/her effort on this iteration of the project.

   Here’s an example. Peter, Paul, and Mary are a 3-member team. In general, they worked well together and followed the expectations and policies that they agreed upon at the beginning of the project. They each took on slightly different roles, but they each contributed roughly equally to the success of the team. In their report, they can simply write their self-assessment as:

Peter	***
Paul	***

Mary	***

Another example: John, Paul, George, and Ringo are a 4-member team. They started off strong and had a great first meeting during Friday's lab. Ringo offered to take the lead on preparing the PDF handin; John, Paul, and George thought this was a great idea because they were eager to spend most of their time learning hands-on C++ programming. They all agreed to meet as a team in person on Tuesday and Thursday nights. The team made some progress in week 1. In week 2, Ringo got busy with other classes and did not show up for the scheduled meetings. He also responded so slowly to emails that he was basically out of the loop for any important programming decisions. At the end, he showed up to try to do his part by creating the PDF handin, but by that point he was so behind that he didn't really have the knowledge to create this document on his own, so John and Paul did it for him. The team discusses the situation, and they divide the stars up as shown below. Ringo is not too happy about this. He was honestly overloaded with responsibilities for other classes, but the other team members point out that he agreed to a plan at the beginning and he let them down. They don't want this to happen again! And, they are obligated to report this via the star system. Ultimately, Ringo can't be too upset because this is essentially a just a warning; however, he knows that he needs to do a better job contributing to the team in the next iteration of the project.

John	****
Paul	****
George	***
Ringo	*

The star system (and the discussion that it forces you to have about how well your team has functioned) is intended to be most useful to you (your group) not us (the instructors). Following the policy in our syllabus, the instructors will use this information simply to get a broad sense of how well each group is functioning as a team. Be honest, this self-assessment is a concrete way for you to critique your group. It will not affect your grade, but if someone in your group is not contributing equally, then this is a very clear way to tell him/her and us.

4. High-Level Grading Rubric

As you plan how to allocate tasks within your group, please keep in mind the following grading rubric that we will use. We include it here specifically to emphasize the importance of the design aspects of this project.

33% - Quality of the writing-related assignments that are part of the Project Release.

33% - Quality of the software design decisions described in the Project Design web page and visible by inspecting the actual source code, Makefiles, and project structure.

33% - Quality of the technical implementation – meets the requirements for PhotoShop, MedImage, and associated testing. Note that part of the quality grade will be based on whether your code compiles on the CSE Labs machines on the first try after we unzip your project source code and run make.

Total: 100%