CSC 412 – Operating Systems

Programming Assignment 06, Fall 2023

Monday, November 13th, 2023

Due date: Sunday, December 3rd, 11:59pm.

1 What this Assignment is About

1.1 Objectives

The objectives of this assignment are for you to

- Getting some experience with the pthread library and the C++ thread class;
- Experiment with the synchronization of threads using mutex locks;
- Learn how to use a third-party library in C/C++;
- Use Doxygen to produce documentation for your code.

This is an individual assignment.

1.2 Handout

The code handout for this assignment is the archive of a C++ project made up of 3 source files and 3 header file, including to the image file input/output code that we have used in the past. The handout also includes a couple of datasets: two "image stacks."

1.3 The problem

In this assignment, you are going to implement a multithreaded version of a "stack focusing" application. You are going to get as input a set of images of the same scene, taken at different focusing distances. Your objective is to produce an output image that combines all the input images so that all parts of the output image are in focus.

2 Part I: Building and Running the Handout

2.1 The code handout

Normally, you shouldn't have to make any changes to the header and source file dealing with the "front end." Even within the `main.cpp` source file, there are clearly identified parts that you should probably don't mess with. Modify any of the "don't touch" sections at your own peril.

2.1.1 About the header files (Doxygen)

I want to attract your attention to a few important points regarding the header files of the handout.

If you are familiar with javadoc style of comments, you probably recognized the characteristic `/**`. For many years, C++ programmers secretly envied their Java-programming colleagues who could produce good-looking code documentation through the javadoc system. They envied them until a brave soul came up with Doxygen, which interprets javadoc-style comments for a multitude of programming languages (C, C++, Python, Perl, etc.).

2.1.2 From now on

In a world where Doxygen exists for all computing platforms, coding in C or C++ is not anymore a good excuse to have a lousy documentation anymore. Starting with this assignment, you will be expected to write javadoc-style comments for all custom data types, global variables, and functions that you define, and to generate documentation in the form of html files. I will return to this point later in this document.

Please note that I won't always remind you that you must provide Doxygen-style comments for all future projects, but there will always be points (typically, about 10) awarded for proper Doxygen documentation.

2.2 OpenGL and glut

2.2.1 What are OpenGL and glut?

This handout, as-is, requires the OpenGL and glut libraries. OpenGL is a library for doing 2D and 3D graphics. OpenGL is blissfully OS-unaware: It has no notion of time. It is equally ignorant of the things called windows, menus, mouse, and keyboard. All it does is render a scene into a buffer. If that buffer happens to be attached to a window, then you will see a rendering of the scene on your display, but OpenGL would be equally happy to render into a buffer attached to the file system or to an output device such as a printer.

Almost every GUI library (including the native GUI library of your OS of choice) provides functions to assign a window's buffer as the place for OpenGL to do its rendering into. Keyboard, mouse, and timer events can be handled either through the GUI's API or through regular system calls. The problem is that native GUI libraries are not portable and that multi-platform libraries (e.g. Qt or wxWidgets) are fairly complex.

This is where glut (which stands for GL utility toolkit) comes in. It was created as a minimalist, very simple library for providing some OS services to OpenGL:

- create and resize windows;
- handle keyboard events;
- handle mouse events;
- handle timer events;
- create drop menus.

A major advantage of glut is that almost exactly the same code can be built and run on different platforms, with only slight cosmetic difference. What differs between different platforms is the location of the headers and binaries for OpenGL and glut. This can be solved by using a header that defines these paths based on the OS and development environment for which the program is built. Some open-source libraries build on top of glut to offer more advanced GUI widgets such as buttons, sliders, file selectors, etc. An example of such a library is pui, which is distributed as part of the plib library.

One minor disadvantage of glut is that the interfaces built with glut (including the ones built on top of glut) are ugly[1]. However, glut was never meant to build commercial-grade applications, just small demos and programming assignments for courses, so this drawback is fairly minor.

The main drawback of glut is that, on Mac and Windows at least, it is still a 32-bit library, now deprecated. It should be replaced by freeglut, which is theoretically 100% compatible with the original glut, the only difference being the name of the header file and binary[2]. On Linux, freeglut has completely replaced glut, so that nothing has to be changed in the code nor in the paths.

2.2.2 About `glut` callback functions

Part of your job is figuring out how much/little you actually need to understand about the operation of the entire program. Don't try to work your way through all parts of the program trying to make sense of what is going on. For example, the actual drawing of Pacman and ghosts are parts that you don't need to bother with. Same with the interface, except for one crucial aspect: the `glut` callback functions and `glutMainLoop`. I will try to explain here briefly the key elements that you need to understand.

The programmer sets up a `glut` interface by defining "callback functions." You essentilly tell glut

- "This is the function to call whenever you need to render the scene" (set by a call to `glutDisplayFunc`);
- "This is the function that you should call whenever the user hits a key on the keyboard" (set by a call to `glutKeyboardFunc`);
- "This is the function to call whenever the user does something with a mouse button" (set by a call to `glutMouseFunc`);
- "This is the function to call if the user tries to change the window's width or height" (set by a call to `glutResizeFunc`);
- "This is the function after a fixed amount of time" (set by a call to `glutTimerFunc`).

After defining all these callback functions, the program relinquishes all control to `glut` by calling the function `glutMainLoop`. It is `glut` that will call the drawing function, will test the keyboard, etc., and will call the callback functions. This works generally well, but threads add a layer of complexity:

[1] This may not be obvious to Linux users, because they don't particularly stand out as being significantly worse than other Linux apps, but on Mac or Windows, the difference is striking.

[2] This being said, in recent versions of mac OS, i have had serious problems with freeglut, to the point that I have decided to revert to Apple's deprecated version of GLUT).

- It is essential that the thread that calls `glutMainLoop` be the main thread. Nothing will work otherwise.
- You must create your threads before the call to `glutMainLoop`.
- Nothing will get drawn (in fact, on some platforms, the window may even not appear) before the call to `glutMainLoop`.
- When the main thread is blocked on a lock, the display will not refresh, and the application will not react to keyboard and mouse events.

2.2.3 Install MESA and FreeGlut on Ubuntu

If you are on macOS, then you don't need to install anything: OpenGL and glut are part of the system (although they are officially tagged as "deprecated" since Apple is trying top push developers to use its new-ish, much lower-level 3D framework named Metal.

On Linux, we are going to use MESA, which is an open-source implementation of OpenGL, the only free option on Linux. MESA and FreeGLUT come installed by default on Ubuntu, so that you can run pre-built OpenGL, but in order to build a MESA project, you need to install the *developer* version of both libraries.
Make sure first that your Ubuntu install is up to date, and then execute:

- `sudo apt-get install freeglut3-dev`
- `sudo apt-get install mesa-common-dev`

2.2.4 How to build the handout

If you are on Linux, then you should be able to build the handout as follows:

```
g++ -Wall -std=c++20 *.cpp -lGL -lglut -o focus
```

Once you start adding threading code to the handout, you will need to link with the `pthread` library. So, the build command will become

```
g++ -Wall -std=c++20 main.cpp gl_frontEnd.cpp -lGL -lglut -lpthread
-o focus
```

On macOS, OpenGl and glut come as *frameworks* (which is the way Apple organizes individual libraries into a single folder containing the dll, static library, and the header files). Additionally, OpenGL and glut are officially deprecated on macOS, which results in a bunch of distracting warnings that it's convenient to disable (these are not telling you anything useful). So the build command is slightly different:

```
g++ -Wall -Wno-deprecated -std=c++20 *.cpp -framework OpenGL
-framework GLUT -o cell
```

Of course, feel free to change the name of the executable. What you cannot change, though, is the order that has you list the libraries to load *after* you have listed the source files to compile. This doesn't make much sense, but this is the way you must do it if you want to avoid linker errors.

2.3 The GUI

When the program runs, you should see a window with two panes. The one on the left displays an image. The right pane shows the state of the computation: The number of "live" focusing threads and some other information that you can/should change if you need to.

There is only one keyboard command enabled: Hitting the 'escape' key terminates the program. If you want to add more keyboard controls, you can edit the function `keyboardCB` in the `main.c` source file.

2.4 Looking into the source files

As mentioned earlier, `imageIO_TGa.cpp` and `imageIO_TGa.h` are nearly the same files that we have used before when we had to read and write images. I have simply done a bit of cleanup and extension of the image data type. In particular, I have moved some code in `rasterImage.cpp` and companion header `rasterImage.h`, but really these are minor revisions that you should (and really ought to) have no problem adapting to.

The files `gl_frontEnd.cpp` and `gl_frontEnd.h` contain the source code and headers necessary to set up the GUI and keep it running. You shouldn't have to change anything in either file, but you are welcome to do it if you [believe that you] know what you're doing.

Finally, there is the `main.cpp` source file. This is the one you should edit and add to. Some parts have been marked **don't touch**. These includes the sections where I define variables used by the front end and make function calls to setup the interface. Changing any of that will break the GUI.

2.5 Threads and GLUT

You will need to rewrite the part around and after Line 250 to load the stack of images and launch the "focusing threads."

You have to be aware of several important things. First, GLUT likes to run on the main thread of the process. The idea is that you set up the GUI, then create processing (for us, focusing) threads, and then give over the control of the main thread to GLUT with the call `glutMainLoop()`. If your program is single-threaded, this opens up the question of who is going to do any work if you hand over the control to GLUT? There are essentially 2 ways of doing that

1. GLUT will call periodically the function `displayImagePane` (it will do so because I set it up with a "timer" function that gets called periodically). All you have to do is do some calculations (or rather, make a call to a function doing calculations) in that function. This is exactly what I do in this handout.

2. The alternative is not to hand over the control to GLUT. This means that you disable the call to `glutMainLoop` and then do your calculations as with a classical headless program. If you want to get a refresh of the display, simply make every now and then a call to `drawImagePane` and `drawStatePane`.

Once you go multithreaded, though, you should definitely hand over the control of the main thread to GLUT by re-enabling that call to `glutMainLoop`.

3 Part II: Introduction to Stack Focusing

3.1 The stack of images

A dataset consists of a series of images of the same scene, taken from the same point and for the same aperture and shutter speed settings, but for different focusing distance settings. If you look at all the images of a same sequence (Figure 1), you can see that, as the focusing distance changes, different parts of the image come into, then out of focus. For any part of the image, there is an image of the stack where that part is "most in focus." The purpose of our application is to produce an output image that is in "best focus" at every single of its points.

Figure 1: A stack of differently-focused images of a scene.

3.2 Focus stack

Let us consider a small rectangular window of our image. Since we have multiple images, what this window defines is really a small volume of pixels, a small rectangular cylinder of pixels that "punches" through our stack of images. we can look a bit closer at our data within this window and see that different parts of the small windows are in focus as the focus distance changes.

3.3 Focus detection

When can we consider that a small region is more "in focus" than another? Obviously, this is a very complex question. As a matter of fact, Canon, Sony, Panasonic, and other big companies spend fortunes each year trying to fine-tune their algorithms. So, we are not trying to solve the problem in what is not even the main focus of this assignment. Still, you are probably going to

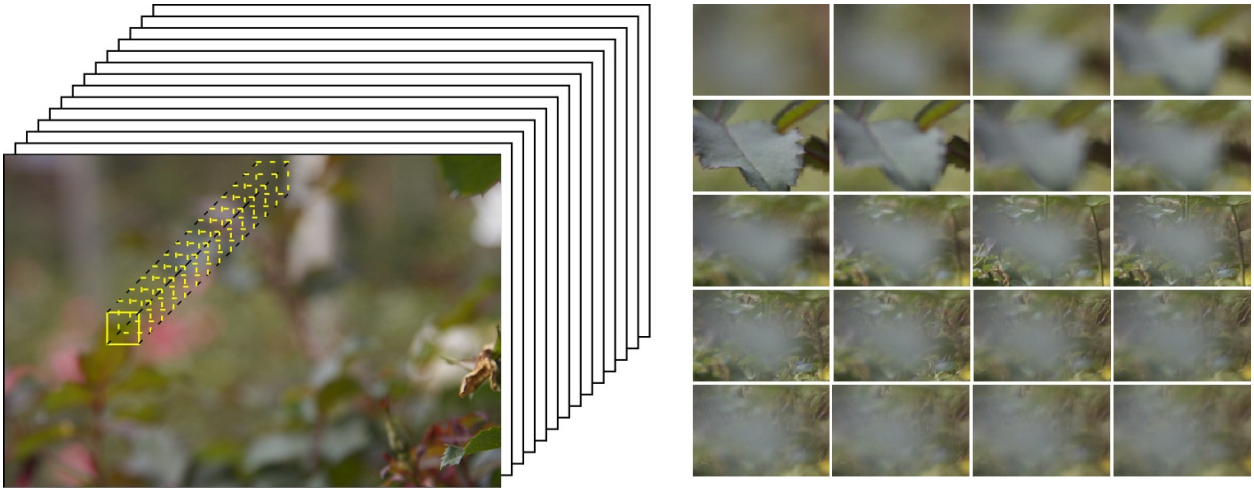

Figure 2: A small rectangular cylinder punches through a stack of images and gives us a series of small rectangular windows at the same location.

be pleasantly surprised by the result. Among the best simple techniques are the ones based on *intensity histograms*. An histogram simply counts the number of times that a particular value is encountered. So, in the case of the black and white boat picture of Figure 3.3, values can take the range [0,255], and the chart of the histogram shows us how frequent each image intensity value is. We observe that the shape of the histogram changes as the image gets blurred.

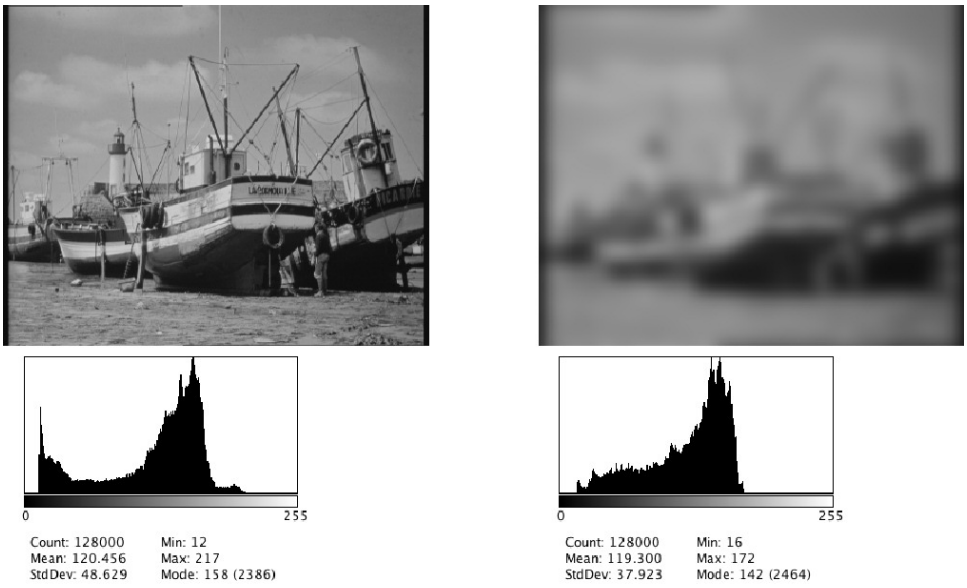

Figure 3: Left: An image and its intensity histogram. Right: A blurred version of this image and the corresponding histogram.

Applying higher levels of blurring only accentuates the changes of the histogram (Figure 3.3).

If you know some fancy statistics, you could start analyzing what is happening to the histogram

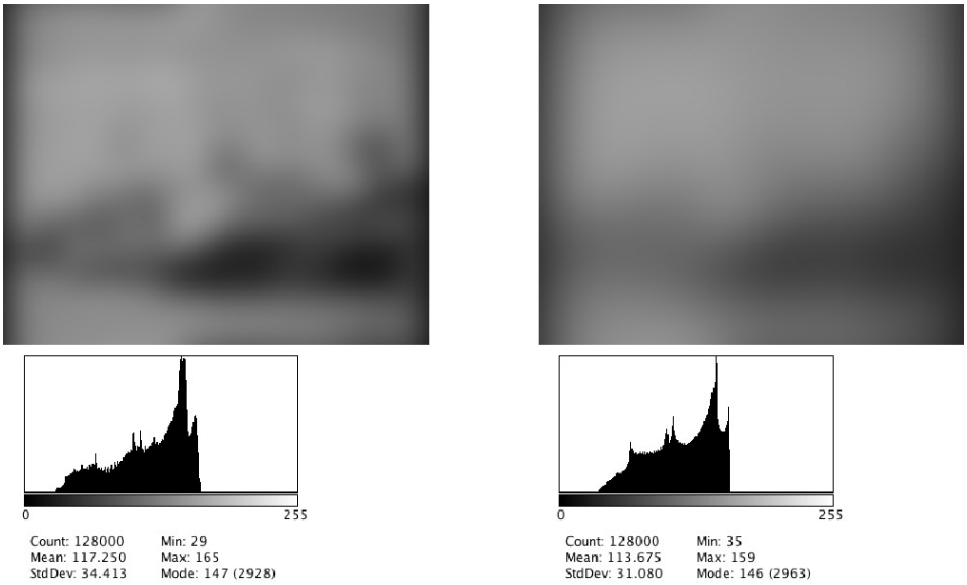

Figure 4: Higher levels of blurring applied to the same image.

and develop an algorithm based on this algorithm. Here, we are going to keep it simple and observe that, as the degree of blur increases, the minimum value observed gets higher, while the maximum value observed gets lower. This makes sense, since a perfectly blurred image would be a uniform rectangle of some shade of gray.

This gives us a very simple "measure" of focus: The difference between the max and min values over an image window: The higher the difference, the more in focus the image window is[3].

3.4 Practical considerations

How big/small a window should we be talking about here? If the window is too small, then we simply haven't got enough samples for our "focus measure" to be correct. If the window is too large, then different parts of the window may be in focus at difference focusing distances and there would be multiple valid candidates for the "most in focus" title. Window sizes such as 5×5 or 7×7 are probably best for this problem.

What about color? The images from the datasets are all color images, so how will the max-min criterion apply? You could decide either to apply the criterion to the gray-converted values at each pixel[4] compute a separate focus measure for each of the three color channels.

[3] As simple as this algorithm is, it constitutes the foundation of what photographers know as "contrast-detection autofocus" (granted, with a lot of optimizations and improvements), of which Panasonic is the last remaining proponent. The drawbacks of the algorithm are mostly encountered when shooting videos (because the autofocus has to wobble around the final focusing distance) and fast-moving targets, but it works quite well for most other still photography projects.

[4] Classical conversions are $gray = red + green + blue$ or $gray = \max(red, green, blue)$.

4 Part III: Implementation

4.1 pthread and C++ thread versions

For each required version of the program, you are going to develop to subversions of it: One using the `pthread` library and one using the C++ `thread` and `lock` classes. Your code will be organized into two folders: `pthread` and one named `C++_thread`. Each will contain three subfolders named `Version1`, `Version2`, and `Version3` containing your implementation of the corresponding version of the program.

4.2 Argument list

Your program takes as arguments:

- the number of computation threads to work over the image stack;
- the path to the output image to write;
- a list of paths to the images making up the image stack.

this means that the paths to the images of the stack make up indices 3 to argc-1 of the argument list. For example, of a small stack made up of 3 images, to be processed by 12 threads, the program could be launched by

```
./focus 12 ./out.tga ./Data/img1.tga ./Data/img2.tga ./Data/img3.tga
```

4.3 Version 1: Multithreaded with no synchronization

This one is very similar to what you did for the gray conversion lab. It simply consists in dividing the image into rectangular regions and assign a thread to each region. Each thread then executes the following algorithm:

for *each pixel in the rectangular region* **do**
  form a small rectangular window centered at the pixel;
  for *each image of the focus stack* **do**
    compute the focus measure over the small window;
  end
  get the color value at the pixel's location for the most in-focus image of the stack;
  assign this color at the pixel's location in the output image
end

Be careful how you handle pixels near the border of the image, as the window centered at the pixel will go out of the image's dimensions.

4.3.1 Extra Credit (up to 6 pts)

Compare the execution time for different numbers of threads (number of rectangular regions into which you divide the image). Write your results and observations in the report.

4.4 Version 2: Multithreaded with a single lock

This version is going to use random sampling to compute the "in focus" output image. Also, instead of writing the value of a single pixel in the output image, it will write values for the entire window centered at the pixel. The basic algorithm for each thread is

```
while still going do
    select a random pixel location;
    form a not-so-small rectangular window centered at the pixel;
    for each image of the focus stack do
        compute the focus measure over the window;
    end
    combine the colors of the most in-focus window of the stack with that in the output
     image;
end
```

For this algorithm, I would like the rectangular windows to be somewhat larger than earlier, maybe 11×11 or even 13×13. The "color combination" of the algorithm is very simple:

```
for each pixel in the most in focus window do
    if pixel value at that location in the output image is 0 (0xFF000000) then
        replace the value at that location by that in the most in focus window;
    else
        for each color channel, r, g, b, of the pixel do
            the new color value is 0.5 * most in focus value + 0.5 * old value in output
             image;
        end
    end
end
```

Synchronization is guaranteed by a single mutex lock for the entire output image. Obviously, the mutex lock should only be used to control the writing into the output image. Multiple threads could be reading the same data with no ill effect.

4.4.1 Extra Credit (up to 5 pts)

Compare the execution time for different numbers of threads (number of rectangular regions into which you divide the image). Write your results and observations in the report.

4.5 Version 3: Multithreaded with multiple locks

The algorithm is the same as in the previous version. The difference is that we are going to divide again the image into rectangular region, and assign a different mutex lock to each region. Before writing a new pixel value into the output image, the thread should make sure that it has obtained the right to write into the region that this pixel belongs to. The problem, as we saw in class, is that

a small image window may intersect with multiple regions, raising the possibility of deadlocks. We also discussed how to address this.

4.5.1 Extra Credit (up to 6 pts)

Compare the execution time for different numbers of threads and different numbers of locks. Write your results and observations in the report.

5 Scripting

We go easy on the scripting with this assignment. All your scripts should go in a folder named `Scripts` located at the root of the project folder.

5.1 Build script

This script, named `build.sh` detect the execution platform (Linux Ubuntu or macOS) and builds all completed versions of the program.

5.1.1 Extra Credit (3 pts)

Provide a script named `script06.sh` that takes as arguments

1. the path to an executable of the stack focusing program,
2. a desired number of threads,
3. the path to the output image to write,
4. the path to a folder containing the images of the stack,

and launches the stack focusing programs with a proper list of arguments.

6 What to Hand In

6.1 Organization of the submission

Submit a zipped archive of a folder containing:

- A directory named `Programs` containing two subfolders named `pthread` and `C++_thread`, each containing three subfolders named `Version1`, `Version2`, and `Version3`, as discussed earlier;
- A directory named `Scripts` containing the build script (and possibly the EC script);
- a directory named `Documentation` containing the html documentation produced by Doxygen;
- your report as a `.pdf` file.

6.2 The report

The report should discuss your design decisions. If you had to interpret some of the specifications of the assignment then state clearly what you did (definitely so if you decided to only implement a subset of the requirements).

Identify any limitations of your programs (besides the obvious, e.g. the fact that you can only handle TGA image files). Are there combinations of arguments that cause your program to crash? Did you decide to only implement a subset of the requirements?

Finally, if you implemented any of the "extra credit" sections, make sure to point it out, so that the graders know what to look for.