CS 2110 Homework 9

Vivian De Sa Thiebaut, Jarrett Schultz, Cem Gokmen, Austin Adams

Fall 2018

Contents

1 Overview **2**

2 Instructions **2**

2.1 Setting Up . . . 2
2.2 Getting Started . . . 2
2.3 Understanding How Pixel Arrays and 16-bit Colors Work . . . 2
2.3.1 How does a 1-d pixel array work for displaying colors on a screen? . . . 2
2.3.2 How are the individual pixels represented as integers? . . . 3
2.4 Drawing Geometry . . . 3
2.5 Color Filters . . . 3

3 Testing Your Work **4**

3.1 Sandbox . . . 4
3.2 Tester . . . 4
3.3 Makefile . . . 4

4 Rubric **5**

5 Deliverables **5**

6 Rules and Regulations **6**

6.1 General Rules . . . 6
6.2 Submission Conventions . . . 6
6.3 Submission Guidelines . . . 6
6.4 Syllabus Excerpt on Academic Misconduct . . . 7
6.5 Is collaboration allowed? . . . 7

1 Overview

For this assignment, you will write in the C language a graphics library that is capable of drawing a variety of shapes, as well as a number of filters that can be applied on the colors of the pixels of an image.

2 Instructions

2.1 Setting Up

It is expected that you work on this assignment and run your code on a computer or VM running Ubuntu 16.04. Run the following commands to install dependencies used to build and test your code:

```
sudo apt update
sudo apt install check build-essential pkg-config valgrind gdb
```

2.2 Getting Started

Read through the *geometry.h* file. The comments here will explain each of the structs so that you know what you're doing when you start writing your functions. This file is thoroughly commented in order to make sure you are able to understand the structs here.

Now take a look at the *graphics.h* file. This file contains the headers and details of each of the functions you need to implement.

You will write all of your code in the *graphics.c* file. If you edit other files, the tester will use their originals and thus your code might not compile. This file is prepopulated with all of the functions you will have to write. If you accidentally delete any, you can still find the prototypes in the *graphics.h* file. Note that for this assignment, some C standard library headers are included, and you can feel free to use any functions included in those headers.

The file is distributed by default with the UNUSED macro in each function that keeps gcc from complaining about unused variables. Without those, you wouldn't be able to compile the file before fully writing all of the functions. So as you are writing each function, **be sure to remove the UNUSED macro calls from its body first**, so that the compiler can accurately warn you about unused variables.

2.3 Understanding How Pixel Arrays and 16-bit Colors Work

2.3.1 How does a 1-d pixel array work for displaying colors on a screen?

At first it might seem weird to represent a 2-d object (the screen) with a 1-d array of pixels, but it's actually pretty straightforward. Think about a text document. If you're typing on one line, eventually it will wrap around to the start of the next line, even though you were just typing one character after the next. The pixel array for the screen works the same way: the end of each row wraps around to the start of the next row. So if the screen is 240x160, for example, then pixels 0-239 are the first row, pixels 240-479 are the second row, continuing for all 160 rows. This convention for storing an array is referred to as "row major" order and is used by most languages, including C, for storing two-dimensional arrays in a one-dimensional memory.

How the pixels look:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

How the pixels are stored:

0	1	2	3	4	5	6	7	8	9	·	·	·		

Figure 1: Visual Explanation of Pixel Layout

2.3.2 How are the individual pixels represented as integers?

We represent each pixel as 16 **unsigned** bits (2 bytes) in a BGR format. With 5 bits to represent the amount of blue, 5 bits for the amount of green, and 5 bits for the amount of red. However, this only adds up to 15 bits, so the most significant bit is unused. The reason it's a BGR format instead of the more commonly known RGB format is because the 5 bits for each color are organized (from most significant bits to least significant bits) blue, green, then red. This is also the layout of pixels on the GBA that we'll use later on.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	Blue					Green					Red				

Figure 2: The layout of bits in a 16-bit BGR format

2.4 Drawing Geometry

You need to implement a number of functions in *graphics.c* that will be used to draw the shapes in *geometry.h*. Prototypes for each function are provided in *graphics.h* in the order that we recommend that you implement them. The documentation above each function explains in detail the expected behavior for that function.

2.5 Color Filters

The filters that you are expected to implement are very simple functions of single pixels that produce a new pixel based on the input. For example, the Greyscale filter will produce a greyscale version of the given pixel, the Red Only filter will only let the red channel of the pixel pass through, and the provided No Filter filter is the identity function on the pixel, so that you can use the drawImage function without implementing the filters first. These functions are also in the *graphics.h* and *graphics.c* files for you to work on.

3 Testing Your Work

You are provided with two different pieces of software to test your work with: the sandbox and the tester.

3.1 Sandbox

The sandbox is provided in the assignment directory. In the *sandbox.c* file there is a bunch of code that sets up a test image and provides you with an environment to test your code with. You can change the code between the comments indicating the area you can change. Once you save the file and run the `make run-sandbox` command from the command line, the resulting image will be output to the *output.bmp* file which you can view using an image viewer.

To help you see cases where you might be writing outside the image bounds, the image will be drawn such that your actual drawing will appear between two horizontal black lines. If you cannot see these lines or if they are incomplete, it means your functions are drawing outside the bounds of the pixel array, which is something you should correct.

The sandbox is shipped with sample images (azul, tvtester, austinbear) that you can use to test your *drawImage* function, too.

The sandbox will not be used for grading, its purpose is to simply show you what image your code draws on a hypothetical screen, in order to make your code easier to debug.

If you need to debug code you wrote in the sandbox, you can run the `make run-sandbox-gdb` command to launch gdb for the sandbox.

3.2 Tester

The tester is provided in the *tester* directory. It contains unit tests that call your functions with certain parameters and compare the output with the TA solution's outputs (the source code for the tests is in *graphics_suite.c*.

The more important feature of the tester is that it shows you as an image file both the output your code produced and the expected output for the tests. Your code's results are found in the *tester/actual/* directory, while the expected images are found in the *tester/expected/* directory, so that you can compare them using any image viewer. Moreover, the *tester/diff/* directory contains images that are diffs of the expected and the actual: pixels that match on the two are marked green and pixels that don't match are marked red.

This is your best bet at debugging your code, and at understanding what's expected of each function! The tester will have the expected image and the image your code produced for all of the test cases. Comparing those should be able to tell you what's wrong with your code.

To run the tester, run the following command (or any of the other ones listed below in the Makefile section:

```
make run-tests
```

Note the tests are unweighted when you run them like this, but they are weighted on Gradescope.

3.3 Makefile

We have provided a Makefile for this assignment that will build your project.
Here are the commands you should be using with this Makefile:

1. To clean your working directory (use this command instead of manually deleting the .o files): `make clean`

2. To run the sandbox: `make run-sandbox`
3. To run the sandbox with gdb: `make run-sandbox-gdb` (gdb instructions are available below, in the HW8 doc, and on Piazza)
4. To run the tests without valgrind or gdb: `make run-tests`
5. To run your tests with valgrind: `make run-valgrind`
6. To debug a specific test with valgrind: `make TEST=test_name run-valgrind`
7. To debug a specific test using gdb: `make TEST=test_name run-gdb`

   Then, at the `(gdb)` prompt:

   (a) Set some breakpoints (if you need to — for stepping through your code you would, but you wouldn't if you just want to see where your code is segfaulting) with `b tester/graphics_suite.c:420`, or `b graphics.c:69`, or wherever you want to set a breakpoint

   (b) Run the test with `run`

   (c) If you set breakpoints: you can step line-by-line (including into function calls) with `s` or step over function calls with `n`

   (d) If your code segfaults, you can run `bt` to see a stack trace

   For more information on gdb, please see one of the many tutorials linked above.

To get an individual test name, you can look at the output produced by the tester. For example, the following failed test is `test_list_size_empty`:

```
suites/list_suite.c:906:F:test_list_size_empty:test_list_size_empty:0: Assertion failed...
                         ^^^^^^^^^^^^^^^^^^^^
```

Beware that segfaulting tests will show the line number of the last test assertion made before the segfault, not the segfaulting line number itself. This is a limitation of the testing library we use. To see what line in your code (or in the tests) is segfaulting, follow the "To debug a specific test using gdb" instructions above.

4 Rubric

The grading will be based on the accuracy of your library in drawing the given inputs. The output of the tester is a good estimate of your final score, however, we reserve the right to add further tests to verify edge case behavior. As a result, it is a good idea for you to test your code in the sandbox using a variety of inputs and inspecting the outputs manually to check if they follow the requirements.

IMPORTANT! You will still submit this assignment on Gradescope, but you will not be able to see your outputs or the expected ones - you will just know if your code passed or failed the tests. As a result, it's a good idea to use the tester locally for debugging.

5 Deliverables

Please upload the following files to **Gradescope**:

1. `graphics.c`

Be sure to check your score to see if you submitted the right files, as well as your email frequently until the due date of the assignment for any potential updates.

6 Rules and Regulations

6.1 General Rules

1. Starting with the assembly homeworks, any code you write must be meaningfully commented. You should comment your code in terms of the algorithm you are implementing; we all know what each line of code does.

2. Although you may ask TAs for clarification, you are ultimately responsible for what you submit. This means that (in the case of demos) you should come prepared to explain to the TA how any piece of code you submitted works, even if you copied it from the book or read about it on the internet.

3. Please read the assignment in its entirety before asking questions.

4. Please start assignments early, and ask for help early. Do not email us the night the assignment is due with questions.

5. If you find any problems with the assignment it would be greatly appreciated if you reported them to the author (which can be found at the top of the assignment). Announcements will be posted if the assignment changes.

6.2 Submission Conventions

1. All files you submit for assignments in this course should have your name at the top of the file as a comment for any source code file, and somewhere in the file, near the top, for other files unless otherwise noted.

2. When preparing your submission you may either submit the files individually to Canvas/Gradescope or you may submit an archive (zip or tar.gz only please) of the files. You can create an archive by right clicking on files and selecting the appropriate compress option on your system. Both ways (uploading raw files or an archive) are exactly equivalent, so choose whichever is most convenient for you.

3. Do not submit compiled files, that is .class files for Java code and .o files for C code. Only submit the files we ask for in the assignment.

4. Do not submit links to files. The autograder does not understand it, and we will not manually grade assignments submitted this way as it is easy to change the files after the submission period ends.

6.3 Submission Guidelines

1. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency let us know **IN ADVANCE** of the due time supplying documentation (i.e. note from the dean, doctor's note, etc). Extensions will only be granted to those who contact us in advance of the deadline and no extensions will be made after the due date.

2. You are also responsible for ensuring that what you turned in is what you meant to turn in. After submitting you should be sure to download your submission into a brand new folder and test if it works. No excuses if you submit the wrong files, what you turn in is what we grade. In addition, your assignment must be turned in via Canvas/Gradescope. Under no circumstances whatsoever we will accept any email submission of an assignment. Note: if you were granted an extension you will still turn in the assignment over Canvas/Gradescope.

3. There is a 6-hour grace period added to all assignments. You may submit your assignment without penalty up until 11:55PM, or with 25% penalty up until 5:55AM. So what you should take from this is not to start assignments on the last day and plan to submit right at 11:54AM. You alone are responsible for submitting your homework before the grace period begins or ends; neither Canvas/Gradescope, nor

your flaky internet are to blame if you are unable to submit because you banked on your computer working up until 11:54PM. The penalty for submitting during the grace period (25%) or after (no credit) is non-negotiable.

6.4 Syllabus Excerpt on Academic Misconduct

Academic misconduct is taken very seriously in this class. Quizzes, timed labs and the final examination are individual work.

Homework assignments are collaborative, In addition many if not all homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be examined using computer programs to find evidence of unauthorized collaboration.

What is unauthorized collaboration? Each individual programming assignment should be coded by you. You may work with others, but each student should be turning in their own version of the assignment. Submissions that are essentially identical will receive a zero and will be sent to the Dean of Students' Office of Academic Integrity. Submissions that are copies that have been superficially modified to conceal that they are copies are also considered unauthorized collaboration.

You are expressly forbidden to supply a copy of your homework to another student via electronic means. This includes simply e-mailing it to them so they can look at it. If you supply an electronic copy of your homework to another student and they are charged with copying, you will also be charged. This includes storing your code on any site which would allow other parties to obtain your code such as but not limited to public repositories (Github), pastebin, etc. If you would like to use version control, use github.gatech.edu

6.5 Is collaboration allowed?

Collaboration is allowed on a high level, meaning that you may discuss design points and concepts relevant to the homework with your peers, share algorithms and pseudo-code, as well as help each other debug code. What you shouldn't be doing, however, is pair programming where you collaborate with each other on a single instance of the code. Furthermore, sending an electronic copy of your homework to another student for them to look at and figure out what is wrong with their code is not an acceptable way to help them, because it is frequently the case that the recipient will simply modify the code and submit it as their own. Consider instead using a screen-sharing collaboration app, such as `http://webex.gatech.edu/`, to help someone with debugging if you're not in the same room.

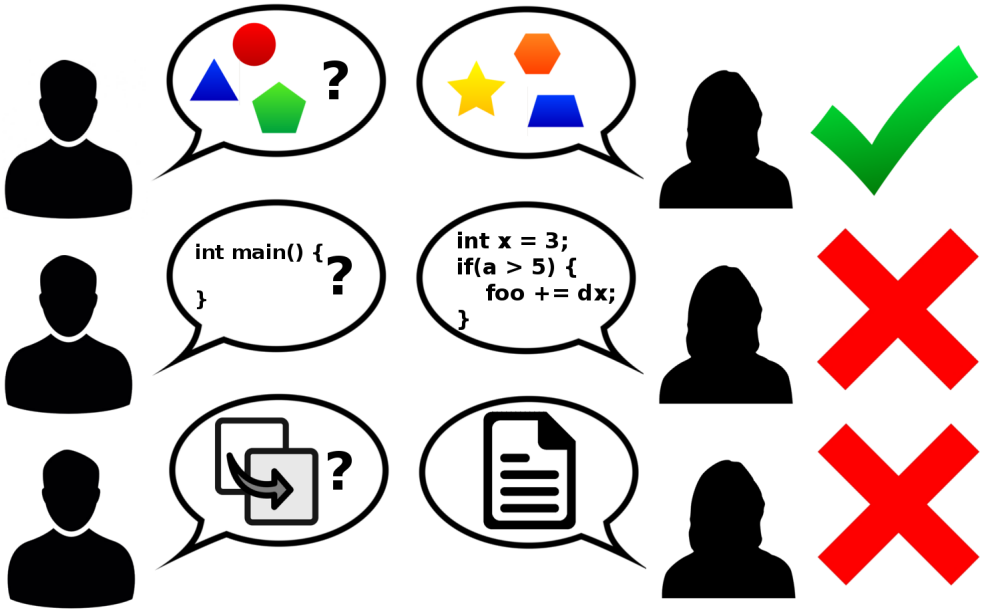

Figure 3: Collaboration rules, explained colorfully