CS 2110 Homework 8
GBA

Jason Ng, Udit Subramanya, John Ever, Jessi Chen

Summer 2022

Contents

1 Overview

1.1 Purpose

The goal of this assignment is to make a C program that will run on a Game Boy Advance emulator. Your program will be an interactive graphical application. Your program can be a game, or any other interactive program that meets the requirements outlined below.

While completing this program, you will learn about low level hardware programming in C. This program is similar to how you would write device drivers or parts of an operating system, which are typically written in C. The GBA devices (screen, buttons, DMA controller, etc.) are accessed via memory-mapped I/O. You will access specific hard-coded memory addresses in your C code. You will gain experience with bit masks and bitwise operators to set bits in the device registers.

The GBA is also a very slow computer. You will learn tricks to optimize the performance on a resource-limited hardware device (such as DMA, integer math instead of floating point, and so forth).

1.2 Tasks

You will write an interactive graphical application in C that runs on the GBA emulator. This application can be a game, or any application that meets the feature and technical requirements below. For ideas on what you could do, please see What to Make in the Appendix. This is an open-ended assignment. Just be sure that your program meets all the requirements.

We have provided some resources to help you get started. You can use the outline structure of a program, and fill in code to do what your application specifically does. You should have at least one `.h` header file and one `.c` file. You can have more if you wish.

Your program must use DMA, `waitForVblank()`, and Mode 3 video. You must have a title screen, in addition to the other screen(s) in your program. You must use button input and include some text on the screen, as well as other images (a background image, and small image, e.g. a character that can move up, down, left, and right). You must detect collisions between two objects on the screen. You must not have any tearing (visual distortion, due to drawing too many things in each screen frame on a slow computer). Please see the full list of requirements below.

1.3 Criteria

You will be evaluated on meeting all the feature requirements and technical requirements described below.

This assignment offers an opportunity for you to be creative if you want to. However, creativity (and artistic ability) are NOT required. Your grade will be based on meeting all the requirements outlined below.

If you write a very creative program, but do not meet all the requirements, you will lose points. On the other hand, if you write a minimal program that works and meets all the requirements, but is not exciting, you will get full credit. Please do not feel pressure to be creative if that is not your personality, or if you do not have time to do so.

2 Before You Start

2.1 Resources

To tackle this homework, we've provided:

- A `gba.h` file that contains all of the necessary GBA declarations such as DMA, `videoBuffer`, etc.
- Several other files which contain more "starter" code to get you rolling. See Section 5.2 for details.
- A Makefile that you can use to compile and run your program by typing or `make mgba` from the docker terminal. *Note:* since there are graphics involved with this assignment, the integrated terminal will not be able to display your project. Make sure to use docker through your browser to view the GBA.

In addition, here are some other helpful resources:

- Lab Guides/Slides
- Lecture Slides 17 and 18
- TONC - full documentation of the GBA, including advanced features
- Lecture Code: Canvas > Files > Source Code > gba_
- The Appendix, including GBA Coding Guidelines

Feel free to use code from class resources as you need to, but as always, not from your friends or random sketchy Internet sites.

Your `main.c` should be something different from lecture code, since in this homework you will be creating your own program. You should keep the core setup with `videoBuffer`, `MODE 3`, `waitForVBlank`, etc., though.

Additionally, please do not rehash lecture code in your program; programs that are merely slightly modified lecture code are subject to heavy penalties. You may borrow some idea and structure from lecture code, but your code should be your own, and your idea should be distinct from lecture code in some way. For example, you are allowed to create games or programs that include bouncing squares, like Pong; however, you should attempt to add new features, like drawing an image instead of a square, or making the square change velocity whenever it bounces. If you have concerns over whether or not your idea is different enough from the s in lecture, feel free to ask a TA in office hours or on Ed Discussion.

2.2 Words of Warning

- Do not use floats or doubles in your code. Doing so will slow your code down greatly. The ARM7 processor the GBA uses does not have a floating-point unit which means floating point operations are slow as they are done in software, not hardware. Anywhere you use floats, `gcc` has to insert assembly code to convert integers to that format. If you do need such things that you think requires floats or doubles, you should look into fixed point math.
- We strongly recommend that you do not use `malloc()` in your program. Instead, use arrays large enough to hold all possible elements (images, game characters, etc.) that you could possibly have at one time. You can use an array of structs, as in the lecture code example.
- Only call `waitForVBlank` once per iteration of your main loop. Each `waitForVBlank` call will stall your program for about one sixtieth of a second, so excessive calls will make your program feel slower.
- Keep your code efficient; particularly, try to do as little drawing per frame as possible. See the guide on Reducing Tearing for tips on how to make your drawing more efficient.

- If your program does have tearing, keep in mind that it will affect the top of the screen first. So, if text or images drawn at the top of the screen are disappearing for some unknown reason, it is likely that your program is attempting to draw during the VDraw period, which is often leads to tearing.
- If you choose to use more advanced GBA features like sprites or sound, making them work is your responsibility; we (the TAs) do not really know how they work, so we sadly can't help you. (TONC is useful to reference when implementing these features, if you do choose to add them.) Note that sound support is currently unstable and is unlikely to work in the current version of the Docker container.

3 Requirements

The autograder for this assignment only checks that your code compiles, so the rest of your grade entirely depends on the following requirements.

Note: the following point values may be changed, but we will not add more requirements or make requirements more strict.

3.1 Feature Requirements

- Your program must compile. This is the most important requirement, since we can't grade your feature requirements otherwise. **(1 point)** *Note:* it is an automatic "0" if your program does not compile.
- You must use **3 distinct images** in your program, all drawn with DMA. **(10 points)**
  - Two full screened images sized 240x160. One of these images should be the first screen displayed when launching your program.
  - A third image which will be used during the course of your program. The width of this image must be less than 240 pixels and the height of this image must be less than 160 pixels.
  - Note: all images should be included in your submission.
  - You should use `nin10kit` (as mentioned in section 5.2.2) to convert images in standard formats like PNG and JPEG into C header files with a constant short array containing 16-bit pixel values. See the Images section in the appendix for instructions on how to create your own custom images.
- You must be able to **reset the program** to the title screen AT ANY TIME using the "select" (backspace) key. This resets the ENTIRE program, including application state. **(10 points)**
- **Button input** should visibly and clearly affect the flow of the program. Examples include pressing the start button or enter key to transition from a game's title screen into gameplay, or using the arrow keys to move an image across the screen. **(10 points)**
- You must have **2-dimensional movement** of at least one entity (an entity that moves both left/right **and** up/down). One entity moving up/down and another moving left/right alone does not count. **(10 points)**
- You should implement some form of **object collision**. Think of detecting collisions as determining if two on-screen items are overlapping. For example, if your player overlaps with a rectangle on the screen, then that is an example of a *collision*. Please make your collision more intricate than an object colliding with the borders of the screen; try to implement collision between objects. For programs where application of this rule is more of a gray area (like Minesweeper), core functionality can take the place of this criteria, such as the numbers for Minesweeper tiles calculated correctly, accurate control, etc. When in doubt, ask a TA for clarification. **(10 points)**
- Use **text** to show progression in your program. **(10 points)**

  - Examples of text progression include on-screen timers, score counters, or any other text that displays useful information to the user.
  - Several functions for drawing strings are already implemented in `gba.h` and `gba.c`. To use these, you must implement `drawPixel` in `gba.c`.
  - You may find the `snprintf` function to be useful when displaying numerical values. This function works like `printf`, except it fills a character array instead of printing to console.
  - See the lecture code for an example of how to implement text progression.
  - If you choose to implement a timer, think about how you can keep track of time. (Hint: the GBA runs at 60 ZFPS, so how can you turn the `vBlankCounter` into "time elapsed"?)

- There must be **no tearing** in your program. Make your code as efficient as possible! Check out the appendix for tips on how to reduce tearing. **(18 points)**

3.2 Technical Requirements

- Include a **README.txt** or **README.md** file with your submission that briefly explains the program and the controls. **Don't forget to put this in your Gradescope submission! (2 points)**
- Your program must be in **Mode 3**! – Mode 3 should be the very first thing set in the main method. (No added points, but this is pretty much required for some of the other requirements.)
- You must also implement drawImage with **DMA**. The prototype and explanation are later in the assignment. Depending on when you are reading this, DMA may not have been covered in lecture yet. If this is the case, you should implement this function with setPixel first, and reimplement it with DMA once it's been covered in lecture. **(6 points)**
- You must implement `waitforVBlank()` **(4 points)**
- You must use at least one **struct**. The struct must be defined in a header (`.h`) file, not a source (`.c`) file. If your program has an object that moves around the screen, you may find it useful to create a struct that stores the object's position. **(5 points)**
- You can create your own **header file** or use the `main.h` header already available. You must move any `#defines`, function prototypes, and typedefs to this file from your code, along with your extern `videoBuffer` statement if you wish to use `videoBuffer` in other files. Remember that function and variable definitions should not go in header files, just prototypes, extern variable declarations, and struct declarations. **(4 points)**
  - As always, do not include `.c` files into other files. Only `.h` files should be included and `.h` files should contain no functional code.
  - It is optional for you to use other `.c`/`.h` files to organize your logic if you wish. Just make sure you include them in your submission and Makefile.

4 Deliverables

Please archive all of your source code files as a zip or a tar and upload to Gradescope under the "Homework 8" assignment. This includes all `.c` and `.h` files needed for your program to compile and run. Do not submit any compiled files. You can use `make clean` to remove any compiled files, or `make submit` to remove compiled files *and* create a tar archive. This tar archive should also contain your README.

5 Appendix

5.1 Appendix A: What to Make?

You may either create your own interactive graphical application the way you wish it to be as long as it fulfills the requirements, or you can make applications that have been made before using your own code. However, your assignment must be yours entirely and not based on anyone else's code. This also means that you are not allowed to base your program off the code posted from lecture. Programs that are merely slightly modified lecture code are subject to heavy penalties. You may borrow some ideas and structure from lecture code, but you should write your own code, and your application should be distinct in some major way. If you have concerns about whether or not your idea is different enough from lecture code or if your idea would satisfy requirements, feel free to ask a TA. Below are some previous programs that you can create or use as inspiration:

5.1.1 Example Programs

Minimum Viable Product:

- As stated in the Overview, you do not need to make a complex or creative program; it must only fulfill the minimum requirements
- Create a start screen and a win screen that use the two required full-screen DMA images
- Use the arrow keys to move a small DMA image around the screen; store this image's position using a struct
- Transition to the win screen when the player-controlled image touches a goal zone (represented via a colored rectangle or image)
- Use text to show an on-screen timer, and display the total time taken on the win screen

Interactive Storybook:

- Recreate a story from a movie or a book using the GBA
- Use text to narrate what is currently happening in the scene
- Use the controls to advance to the next scene or control a character within the scene
- Smooth movement (for any moving characters or objects)
- Start off with a full screen title image and end with a full screen credits image
- Characters represented by structs

Galaga:

- Use text to show lives
- Game ends when all lives are lost. Level ends when all aliens are gone.
- Different types of aliens: there should be one type of alien that rushes towards the ship and attacks it
- Smooth movement (aliens and player)
- Aliens and the ship represented by structs

The World's Hardest Game:

- Smooth motion for enemies and player (no jumping around)
- Confined to the boundaries of the level
- Enemies moving at different speeds and in different directions
- Sensible, repeating patterns of enemy motion
- Enemies and the player represented by structs

Flyswatter:

- Images of flies moving smoothly across the screen
- Player controlled flyswatter to swat the flies
- Score counter to keep track of how many flies have been swatted
- Fullscreen image for title screen and game background
- Enemies and the player represented by structs

Text Editor:

- Pick some sample pre-rendered text, and display it in a window on the screen.
- Use the up/down/left/right buttons to move a cursor through the text. (Display the cursor somehow. Advanced: use time to make it blink over the current position.)
- Pick another button to be the mouse down button. When the mouse is down, the arrow keys will select the text you mouse over. (Highlight the selected text. Perhaps reverse foreground and background colors.)
- Choose three more buttons to be cut (ctrl-X), copy (ctrl-C), and paste (ctrl-V). Paste will insert text (from your clipboard) at the current cursor position.
- Collision detection (feature requirement) involves determining which character the cursor is pointed at.
- Find a way to include the three required images.

Color Picker:

- There are 32,768 unique colors on the GBA. Draw a two-dimensional color palette rectangle, including all the possible colors.
- Use the arrow keys as a mouse to move a crosshairs cursor around the color palette.
- Choose a key to select a color. Display that color in a solid box somewhere else on the screen.
- Optionally choose a separate background color, and display that. (Find a way to choose if you are selecting foreground or background color.)
- Collision detection - which color am I currently pointing the mouse at?
- Find a way to implement the three required images.
- Variation (or extension of this): Implement part of Microsoft Paint - a simple drawing program. Use the arrow keys as a cursor. Use another key to select draw or erase (or other tools).

5.2 Appendix B: GBA Coding Guidelines

5.2.1 Building and Running your Code

To build your code and run the emulator, run:

```
$ make mgba
```

This must be run inside of the Docker container via noVNC (that is, running in your browser). Since the application is graphical, the emulator will fail to start if you are running via `./cs2110docker.sh -it`.

5.2.2 Images

As a requirement, you must use at least 3 images in your program. To use images in GBA, you will first have to convert them into the suitable format. We recommend using a tool called `nin10kit`, which is pre-installed on the Docker image, or you just installed using the command above.

You can read about `nin10kit` in the nin10kit documentation (there are pictures!):

`https://github.com/TricksterGuy/nin10kit/raw/master/readme.pdf`

`nin10kit` reads in, converts, and exports image files into C arrays in .c/.h files ready to be copied to the GBA video buffer by your implementation of `drawImageDMA()`! It also supports resizing images before they are exported.

You want to use Mode 3 since this assignment requires it, so to convert a picture of smelly festering garbage into GBA pixel format in `garbage.c` and `garbage.h`, resizing it to 50 horizontal by 37 vertical pixels, you would run `nin10kit` **from the same directory as your images** using:

```
$ nin10kit --mode=3 --resize=50x37 garbage garbage.png
```

This creates a `garbage.h` file containing

```
extern const unsigned short garbage[1850];
#define GARBAGE_SIZE 3700
#define GARBAGE_LENGTH 1850
#define GARBAGE_WIDTH 50
#define GARBAGE_HEIGHT 37
```

which you can use in your program by saying `#include "garbage.h"`.

The `garbage.c` generated, which you should add to the Makefile under `OFILES` as `garbage.o` if you plan to use it, contains all of the pixel data in a huge array:

```
const unsigned short garbage[1850] =
{
    0x7fff,0x7fff,0x7fff,0x7fff,0x7fff, // ...
    0x7fff,0x7fff,0x7fff,0x7fff,0x7fff, // ...
    // ...
    0x7fff,0x7fff,0x7fff,0x7fff,0x7fff, // ...
    0x7fff,0x7fff,0x7fff,0x7fff,0x7fff, // ...
};
```

We've included `garbage.png`, `garbage.c`, and `garbage.h` in the homework zip so you can check them out yourself. To draw the garbage in your own program, you can pass the array, width, height to your `drawImageDMA()` like `drawImageDMA(10, 20, GARBAGE_WIDTH, GARBAGE_HEIGHT, garbage)` (to draw at row 10 and col 20). (Keep in mind that this image will only work with drawImageDMA; it is not a fullscreen image.) The next section will cover `drawImageDMA()` in more detail.

5.2.3 DMA / drawImageDMA

In your program, you must use DMA to code drawImageDMA.

The GBA screen is represented with a `short` pointer declared as `videoBuffer` in the `gba.h` file. The pointer represents the first pixel in a 240 by 160 screen that has been flattened into a one dimensional array. Each pixel is a `short` and has red, green, and blue channels.

DMA stands for Direct Memory Access and should be used to make your rendering code run much faster. If you want to read up on DMA before it is covered in lecture, you may read these pages from Tonc. `http://www.coranac.com/tonc/text/dma.htm` (Up until 14.3.2).

If you want to wait, then you can choose to implement drawImageDMA without DMA and then when you learn DMA rewrite it using DMA. Your final answer for drawImageDMA must use DMA.

You must not use DMA to do single-pixel copies (doing so defeats the purpose of DMA and is actually slower than just using setPixel!). Solutions that do this will receive no credit for that function. The prototype and parameters for drawImageDMA are as follows.

```
/* drawImageDMA

* A function that will draw an arbitrary sized image
* onto the screen (with DMA).
* @param row the row coordinate to start drawing the image at
* @param col the col coordinate to start drawing the image at
* @param width width of the image
* @param height height of the image
* @param image pointer to the first element of the image
*/
void drawImageDMA (int row, int col, int width, int height, const u16 *image) {
    // TODO: implement :)
}
```

5.2.4 Other DMA functions

Stubs and comments for other DMA functions are provided in `gba.c`. You are not required to implement them, as they are not graded. However, it is likely that you will need to use at least some of them while developing your application.

Tip #1: if your implementation of a DMA function does not use all the parameters that are passed in then you are not implementing the function correctly.

Tip #2: For implementing DMA functions, you should know that DMA acts as a for loop, but it is done in hardware.

Tip #3: The number of DMA calls needed to implement

5.2.5 GBA Controls

Here is the mapping between GameBoy buttons and keyboard keys in our configuration of the `mgba` emulator:

GameBoy	Keyboard
Start	Enter
Select	Backspace
A	Z
B	X
L	A
R	S

The directional arrows are mapped to the same directional arrows on the keyboard.

5.2.6 C Coding Conventions

- Do not jam all your code into one function (i.e. the main function)
- Split your code into multiple files (for example, you can have your main logic in your main files, but other helper functions for drawing or assessing the application state in external files)
- Do not include .c files into other files. Only .h files should be included.
- .h files should contain no functional code.
- Comment your code, and comment what each function does.

5.2.7 Reducing Tearing

- If you see images at the top of the screen start to flicker, you are likely experiencing tearing. This typically happens because you are drawing during the VDraw phase of the GBA's drawing cycle, which means there is a possibility that any pixels that you write to the video buffer will not be drawn until the next VDraw cycle.
- To reduce tearing, it is first necessary to understand how much you can draw in a single VBlank cycle (the following stats were measured by Brandon Whitehead, a former 2110 TA):
  - Drawing a fullscreen image without DMA will take over three full VBlank cycles. You should never, ever draw this many pixels without using DMA.
  - Drawing a fullscreen image with DMA will take 124 scanlines, which is more than a single VBlank phase! So, even if you start at the very beginning of a VBlank phase, you can never draw an entire fullscreen image without entering VDraw. This is not necessarily bad, since DMA typically starts at the top of the screen, so the top portion of the full screen image is drawn before there is any risk of tearing; however, this also means that any text or images drawn at the top of the screen after the full-screen image will not be drawn until the VBlank period is over.
  - The maximum amount of pixels drawable during a VBlank phase WITHOUT DMA is approximately 1610. Remember this when drawing text, since text does not use DMA.
  - The maximum amount of image pixels drawable during a VBlank phase WITH DMA is approximately 20690. Remember this if you plan to draw many images on-screen.
  - Clearing the screen with a constant color using DMA is barely possible inside a single VBlank phase (63 scanlines). 32-bit DMA can be used to fill the screen in 47 scanlines (approximately 70 percent of the VBlank phase).
- The best way to reduce tearing is to simply draw less. As explained above, using a solid color background is more efficient than an image background. You should also minimize the number of images on-screen, and use text sparingly.
- However, it is possible to use an image as a background by using an **undraw function**. This is a function that takes a full-screen image and draws only a portion of the image to the screen. By using this function, you can skip having to redraw the entire background whenever the player moves; instead, you only need to redraw the portion near the player.
- It may also help to be efficient when "undrawing" images. For example, if the player is a 16x16 image and the player moves one pixel to the right, it is tempting to either redraw the entire background, or at the very least redraw a 16x16 portion of the background where the player used to be. However, you only really need to redraw the 16x1 rectangle sliver that player evacuated when it moved one pixel to the right.

- If you simply must draw a fullscreen image and other small images and text in a single frame, there are a few workarounds. One easy option is to create a second buffer (a 38400-element `short` array) that you will "draw" to during VDraw; then, as soon as VBlank begins, you can DMA the second buffer's data to the main video buffer. A more difficult option is to draw everything at the top of the screen first, then everything at the bottom of the screen; this avoids tearing by taking advantage of the fact that scanlines go from the top of the screen downwards.

5.2.8 Debugging

The GBA does not have a console to print to, so you cannot rely on traditional `printf` debugging. In fact, the GBA doesn't even have an operating system that can handle any calls to `printf`. You should instead use the GNU debugger (`gdb`) to debug your program.

To launch your application in MGBA and start an attached GDB session in your terminal, run the command `make gdb` from the terminal within the Docker container. To exit GDB, use the `quit` command.

Learning how to use GDB will be covered in lab. There is also a supplemental video on the topic here. Additionally, there is a GDB cheat sheet here which describes more than enough commands to be able to effectively debug this homework.

5.2.9 Making Sense of the Files

As mentioned in the C Coding Conventions section, it's often a good idea to split up functionality into multiple files. In fact, this is exactly what we've done with the "starter" code we've given you. The pure volume of files may be a bit daunting, so here's a brief breakdown of what each file is used for.

- `Makefile`

  This Makefile contains all of the tasks you can run to build and test your program. You'll need to modify it to include the `.o` file for any image you'd like to use. However, you should not modify the bottom portion of this file, as bad things may happen. Feel free to look through it in order to see all of the tasks at your disposal.

- `main.c`

  This file contains a **state machine** which ultimately calls all other functionality in the program. It is also the main entry point to the entire application.

- `main.h`

  This file should contain any function prototypes and structs that you create.

- `gba.h`

  This file contains a large collection of useful macros and constants which will help primarily with GBA-specific tasks. These include macros for handling GBA input, DMA graphics, and general GBA graphics.

  The file also contains an extern declaration of the font data found in `font.c`, which is necessary for drawing text.

  This file also contains some prototypes for functions in `gba.c`.

- `gba.c`

  The functions you will write in this file do the "dirty" work, executing graphics updates with both DMA and non-DMA strategies. All of this code will be very specific to the GBA platform and the way it handles graphics.

  The file also comes with some prepackaged functions for drawing text.

- `font.c`

  Simply exists to store a large amount of font data. No real need to mess around with this file.

- Various Image Files

  When you create your own image files, you will need to include the relevant header files in any file you'd like to reference these images from. The data stored in each of these files (and how to create them) is pretty well explained in section 5.3.

- Personal `.c` files and `.h` files

  You are heavily encouraged to split your code up into separate .c and .h files! Do not put everything in your main method. Keep your code organized and readable.

  NOTE: For any `.c` files you create please place its respective `.o` file in the `OFILES` variable in the Makefile. For example, if you create **newFile.c** you will need to update the Makefile to reference the `newFile` object file.

  ```
  OFILES = gba.o font.o main.o images/garbage.o newFile.o
  ```

5.2.10 Submitting

To submit your code:

1. Make sure your code compiles by running `make mgba`
2. Clean the build artifacts by running `make clean`
3. Create the submission tar by running `make submit`
4. Turn in `submission.tar.gz` on Gradescope!

5.3 Appendix C: Rules and Regulations

5.3.1 General Rules

1. Although you may ask TAs for clarification, you are ultimately responsible for what you submit. As such, please start assignments early, and ask for help early. 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.

2. If you find any problems with the assignment it would be greatly appreciated if you reported them to the TAs. Announcements will be posted if the assignment changes.

5.3.2 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 unless instructed otherwise.

5.3.3 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

5.3.4 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.