

CSC 452 – Project 1: Graphics Library

Due: Sunday, September 12, 2021, by 11:59pm

Project Description

Your instructor was learning ARM assembly language over a past summer break and was trying to make some projects. While text-based programs are okay, it's really the graphical programs that make it a bit more interesting. So, your instructor started to write a graphics library so that students could make a simple game.

In doing so, he noticed that a lot of the basic functionality the library provided was accessing the low-level operating system features that we talk about in OS. The light bulb went off over his head, and project 1 was born.

In this project, you'll be writing a small graphics library that can set a pixel to a particular color, draw some basic shapes, and read keypresses.

How it Will Work

You will provide the following library functions (explained further below):

Library Call	System Call(s) used
<code>void init_graphics()</code>	open, ioctl, mmap
<code>void exit_graphics()</code>	ioctl
<code>char getkey()</code>	select, read
<code>void sleep_ms(long ms)</code>	nanosleep
<code>void clear_screen(void *img)</code>	
<code>void draw_pixel(void *img, int x, int y, color_t color)</code>	
<code>void draw_line(void *img, int x1, int y1, int x2, int y2, color_t c)</code>	
<code>void *create_buffer()</code>	mmap
<code>void blit(void *src)</code>	

Each library function must be implemented using only the Linux system calls. You may **not** use any C standard library functions in your library anywhere.

`init_graphics()`

In this function, you should do any work necessary to initialize the graphics library. This will be at least these four things:

1. You will need to open the graphics device. The memory that backs a video display is often called a **framebuffer**. In the version of Linux we are using, the kernel has been built to allow you direct access to the framebuffer. Hardware devices in Unix/Linux are usually exposed to user programs via device “files” in the `/dev/` filesystem. In our case, we will find a file, `/dev/fb0`, that represents the first (zero-th) framebuffer attached to the computer.

Since it appears to be a file, it can be opened using the `open()` syscall. To set a pixel, we only need to do basic file operations. You could seek to a location and then write some data. However, we will have a better way.

2. We can do something special to make writing to the screen easier than using a whole bunch of seeks and writes. Since each of those is a system call on their own, using them to do a lot of graphics operations would be slow. Instead, we can ask the OS to map a file into our address space so that we can treat it as an array and use loads and stores to manipulate the file contents. We will be covering this more later when we discuss memory management, but for now, we want to use this idea of **memory mapping** for our library. The `mmap()` system call takes a file descriptor from `open` (and a bunch of other parameters) and returns a `void *`, an address in our address space that now represents the contents of the file. That means that we can use pointer arithmetic or array subscripting to set each individual pixel, provided we know how many pixels there are and how many bytes of data is associated with each pixel.

One note is that we must use the `MAP_SHARED` parameter to `mmap()` because other parts of the OS want to use the framebuffer too.

3. In order to use the memory mapping we have just established correctly, we need some information about the screen. We need to know the resolution (number of pixels) in the x and y directions as well as the bit-depth: the amount of colors we can use. The machine that QEMU is configured to emulate is in a basic 640x480 mode with 16-bit color. This means that there are 640 pixels in the x direction (0 is leftmost and 639 is rightmost) and 480 pixels in the y direction (0 is topmost, 479 is bottommost). In 16-bit color, we store the intensity of the three primary colors of additive light, Red, Green, and Blue, all packed together into a single 16-bit number. Since 16 isn't evenly divisible by three (RGB), we devote the upper 5 bits to storing red intensity (0-31), the middle 6 bits to store green intensity (0-63), and the low order 5 bits to store blue intensity (0-31).

You can then use `typedef` to make a color type, `color_t`, that is an unsigned 16-bit value. You can also make a macro or function to encode a `color_t` from three RGB values using bit shifting and masking to make a single 16-bit number.

To get the screen size and bits per pixels, we can use a special system call: `ioctl`. This system call is used to query and set parameters for almost any device connected to the system. You pass it a file descriptor and a particular number that represents the request you're making of that

device, plus a pointer to where the result will go (if applicable). We will use two requests: `FBIOGET_VSCREENINFO` and `FBIOGET_FSCREENINFO`. The first will give back (by changing the third parameter – a pointer to) a struct `fb_var_screeninfo` that will give the virtual resolution. The second will do the same for a struct `fb_fix_screeninfo` from which we can determine the bit depth. The total size of the `mmap()`'ed file would be the `yres_virtual` field of the first struct multiplied by the `line_length` field of the second.

4. Our final step is to use the `ioctl` system call to disable keypress echo (displaying the keys as you're typing on the screen automatically) and buffering the keypresses. The commands we will need are `TCGETS` and `TCSETS`. These will yield or take a struct `termios` that describes the current terminal settings. You will want to disable canonical mode by unsetting the `ICANON` bit and disabling `ECHO` by forcing that bit to zero as well.

There is one problem with this. When the terminal settings are changed, the changes last even after the program terminates. That's why we have the `exit_graphics()` call. It can clean up the terminal settings by restoring them to what they were prior to us changing them.

That handles the normal termination, but it's always possible that our program abnormally terminates, i.e., it crashes. In that case when you try to type something at the commandline, it will not show up on the screen. What you're typing is still working, but you cannot see it. You can type the command `stty echo` to turn echo back on.

exit_graphics()

This is your function to undo whatever it is that needs to be cleaned up before the program exits. Many things will be cleaned up automatically if we forget, for instance files will be closed and memory can be unmapped. It's always a good idea to do it yourself with `close()` and `munmap()` though.

What you'll definitely need to do is to make an `ioctl()` to reenale key press echoing and buffering as described above.

getkey()

To make games, we probably want some sort of user input. We will use key press input and we can read a single character using the `read()` system call. However, `read()` is blocking and will cause our program to not draw unless the user has typed something. Instead, we want to know if there is a keypress at all, and if so, read it.

This can be done using the Linux non-blocking system call `select()`.

sleep_ms()

We will use the system call `nanosleep()` to make our program sleep between frames of graphics being drawn. From its name you can guess that it has nanosecond precision, but we don't need that level of granularity. We will instead sleep for a specified number of milliseconds and just multiply that by 1,000,000.

We do not need to worry about the call being interrupted and so the second parameter to `nanosleep()` can be `NULL`.

create_buffer()

If we draw directly to the framebuffer, a complex scene may render slow enough to see each part appear on screen. To get the illusion of more fluid graphics, a technique known as **double buffering** is often used. We make a duplicate array of pixels in RAM rather than representing any hardware device. We draw to this “offscreen buffer” by making it the target of all our graphical operations. And when it’s time to show the completed scene, we copy the offscreen buffer into the framebuffer in one big memory copy operation, known as a **blit** (see the next section).

For the `create_buffer()` function, we will allocate a screen-sized region of our address space. Normally, we’d use the C Standard Library function `malloc()` to do this, but we’re avoiding all library calls. So we will once again ask the OS for memory directly, by using the same `mmap()` system call we used in `init_graphics()`. This time, we will pass slightly different parameters. The memory should still be readable and writeable, but rather than `MAP_SHARED`, we wish to make a private, anonymous allocation. Anonymous means that we do not have a file to map into the region. Use `MAP_PRIVATE | MAP_ANONYMOUS` and a file descriptor of `-1` to use `mmap()` just like `malloc()`. The return value (if successful) will be the address of an appropriately-sized (screen-sized) contiguous block of address space. Return that address back to the caller.

blit()

A blit is simply a memory copy from our offscreen buffer to the framebuffer. If we were using the C Standard Library, we’d do this using `memcpy()`. But we must write it ourselves. So, use one or two `for` loops and copy every byte from the source offscreen buffer onto the frame buffer from the offscreen buffer referred to by `*img`.

draw_pixel()

This is the main drawing code, where the work is actually done. We want to set the pixel at coordinate `(x, y)` to the specified color. You will use the given coordinates to scale the base address of the memory-mapped framebuffer (`img`) using pointer arithmetic. The image will be stored in **row-major order**, meaning that the first row starts at offset 0 and then that is followed by the second row of pixels, and so on.

draw_line()

Using `draw_pixel`, make a line from `(x1, y1)` to `(x2, y2)`. Use Bresenham’s Algorithm with only integer math. Find your favorite online source to do so (and please cite the source in a comment), but make sure your implementation works for all valid coordinates and slopes.

clear_screen()

When we want to blank out our offscreen buffer or our framebuffer, we can just copy over every byte with the value zero. So, much like our `blit()`, loop over the image buffer parameter and set each byte to zero.

Installing and Running QEMU

For everyone

Download the qemu-arm.zip file from the website and extract the files into a folder.

On Windows

Double-click the start.bat file in the folder to launch QEMU.

On Mac OS X

Install QEMU through Homebrew. If you don't have Homebrew, open a terminal and type:

```
/bin/bash -c "$(curl -fsSL  
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Go through the install steps. When done, install qemu by typing:

```
brew install qemu
```

That will install qemu. . Now you can run ./start.sh in a Terminal in the unzipped file directory to launch qemu.

On Linux

Using your appropriate package manager, install qemu-system-arm, likely part of your distro's qemu package.

Then run start.sh in the zipped folder to launch qemu.

Setting up the Development Environment

To keep downloads small, the disk image we have provided does not have a full development environment installed. To install gcc, gdb, and ssh/scp, run the script:

```
./devtools.sh
```

When this finishes downloading and installing, you should have the ability to use most basic Linux commands, nano for text editing, and gcc, gdb, and ssh/scp/sftp for transferring files. These commands will survive a reboot, so this only needs to be done once.

If you want to install other programs, you may use the Tiny Core Linux distribution's package installer:

```
tce
```

This lets you search for a particular package or program name and install it.

After you've finished installing the tools, you may wish to make a backup of the disk image in case something happens and you don't feel like redoing all of these steps.

Shutdown linux using the command (rebooting has been turned into poweroff by an option to QEMU):

```
sudo reboot
```

Then, make a copy of `disk.qcow2` someplace safe. If things go wrong, you can always restore back to this point by replacing the `disk.qcow2` file in this directory with the one you've backed up.

File Backups

I suggest making a directory on Lectura under your home directory that no one else can see.

To do so, do:

```
mkdir private
```

```
chmod 700 private
```

Backup all the files you change under QEMU to your `~/private/` directory frequently!

Loss of work not backed up is not grounds for an extension. **YOU HAVE BEEN WARNED.**

Copying Files In and Out of QEMU

With the dev tools installed you can use `scp` (secure copy) to transfer files in and out of QEMU.

You can download the test driver program from lectura with the command:

```
scp USERNAME@lectura.cs.arizona.edu:~jmisurda/original/hilbert.c.
```

You can backup a file named `library.c` to your private folder with:

```
scp library.c USERNAME@lectura.cs.arizona.edu:private
```

Building

We want to be able to use the functions of `library.c` with multiple programs. So we are keeping the file with `main()` separate from the implementation of our graphics functions.

To build a C program from two source files, place them both on the same `gcc` command line, using the `-o` option to provide a name (otherwise it defaults to producing a program named `a.out`). For example:

```
gcc -o testprogram square.c library.c
```

And then we can run the resulting program with

```
./testprogram
```

Recall that all C functions need to be prototyped prior to their being called the first time. We are requiring a header file which prototypes all of the required functions including the RGB function/macro.

Do not put functions in the header which are “private” and be sure to declare them with static linker visibility so no one can call them outside of your file.

Requirements and Submission

You need to submit:

- Your library.c file containing the implementation of the above functions using pure Linux system calls and no C Standard Library Calls
- A graphics.h header file which defines prototypes for all the “public” functions, your color_t typedef, and your RGB conversion macro.
- An additional driver program of your design that shows the required function calls working

We will use the turnin program on lectura to turn in your project. If your three files are named library.c, graphics.h, and driver.c, execute the following command from lectura in the directory that contains them:

```
turnin csc452-fall2021-p1 library.c graphics.h driver.c
```