Graphics Tutorial

Low-Level Graphics on Linux

Tutorial 1: Intro to Low-Level Graphics on Linux

Introduction

This tutorial attempts to explain a few of the possible methods that exist on Linux to access the graphics hardware from a low level. I am not talking about using Xlib instead of GTK+ or QT5, nor am I talking about using DirectFB, I want to go even lower than that; I'm talking about drawing graphics to the screen without needing any external dependencies; I'm talking about communicating directly with the Linux kernel. I will also provide information about programming for newer graphical systems (Wayland/Mir) even though those do not involve direct communication with the kernel drivers. The reason I want to provide this information in this tutorial is that even though their APIs are higher level, the programming techniques used in low-level graphics programming can easily be adapted to work with Wayland and Mir. Also, similar to fbdev and KMS/DRM APIs, good programming resources are hard to come by.

Most Linux systems actually provide a few different methods for drawing graphics to the screen; there are options. However, the problem is that documentation is basically non-existent. So, I would like to explain here what you need to know to get started.

Please note that this tutorial assumes you have a basic knowledge of C, this is not a beginner tutorial, this is for people who are interested in something like learning more about how Linux works, or about programming for embedded systems, or just doing weird experimental stuff for fun.

Contents

Methods for Rendering in Linux

The most common graphics architecture in Linux by far is X11. However, this is not the only way that Linux has to display graphics. In addition, X11 has some disadvantages which are going to be different for everybody. For now, we are going to say that using xlib or xcb for rendering is too high level. We want to learn about lower level options, preferably communicating only with the Linux kernel. There are a few options available, the two main options are the Linux framebuffer device and DRM. Most other options are not feasible for a single person to do by him/her self. In addition to these kernel options, there are a few upcoming graphical systems which are interesting in the way that they work, being programmed similarly to the kernel interfaces and providing a hopefully easy path to port your low level application into a fully windowed environment, in particular I am talking about freedesktop.org's Wayland, and Ubuntu's Mir. Although these systems are not in use yet, they can be compiled and installed, and you can write applications for them.

Linux Framebuffer Device (fbdev)

The Linux Framebuffer is often talked about, but rarely actually used. One of the main reasons for this is that documentation is fairly hard to come by. Like many things, the people that know how to program for the framebuffer are few and far between, and for some reason, they aren't prone to share their knowledge. However, it is possible and not too difficult to actually make it work. So here's a little explanation of how to render to it.

First of all, the Linux kernel must be built with support for the correct Framebuffer driver. If none is available for your graphics card, you can use the generic VESA driver, but this requires an additional parameter to be passed to the kernel at boot time. Other drivers shouldn't require this.

If your kernel provides the framebuffer device, it will be called fb0 (or fb1, fb2, etc.. if there is more than one active driver) and will be located in /dev. This appears as a file, but in fact it is not a file, it is a file-like interface to the device driver, part of the Unix Everything-is-a-file concept. Although it is not a real file, we can still open it (and read to it and write to it) like a file. This is the very first thing we want to do, open the device file.

```
        int fb_fd = open("/dev/fb0",O_RDWR);
```

Once the file is open, we can actually start writing to it. However, this would not be useful since we don't know the dimensions or color depth of the screen yet, so we would not be able to accurately draw anything meaningful inside the buffer. So the next thing that we should do it get some basic information about the screen. We can do this with the ioctl function (or syscall). There are two structures defined in linux/fb.h for storing info about the screen. They are called fb_var_screeninfo and fb_fix_screeninfo. We should create an instance of each of these structs. (also, remember to include linux/fb.h)

```
#include <linux/fb.h>


...


        struct fb_fix_screeninfo finfo;
        struct fb_var_screeninfo vinfo;
```

These structures are defined in linux/fb.h as follows:

```
struct fb_fix_screeninfo {
        char id[16];                    /* identification string eg "TT Builtin" */
        unsigned long smem_start;       /* Start of frame buffer mem */
                                        /* (physical address) */
        __u32 smem_len;                 /* Length of frame buffer mem */
        __u32 type;                     /* see FB_TYPE_*                */
        __u32 type_aux;                 /* Interleave for interleaved Planes */
        __u32 visual;                   /* see FB_VISUAL_*              */
        __u16 xpanstep;                 /* zero if no hardware panning  */
        __u16 ypanstep;                 /* zero if no hardware panning  */
        __u16 ywrapstep;                /* zero if no hardware ywrap    */
        __u32 line_length;              /* length of a line in bytes    */
        unsigned long mmio_start;       /* Start of Memory Mapped I/O   */
                                        /* (physical address) */
        __u32 mmio_len;                 /* Length of Memory Mapped I/O  */
        __u32 accel;                    /* Indicate to driver which     */
                                        /*  specific chip/card we have  */
        __u16 capabilities;             /* see FB_CAP_*                 */
        __u16 reserved[2];              /* Reserved for future compatibility */
};

...

struct fb_var_screeninfo {
        __u32 xres;                     /* visible resolution           */
        __u32 yres;
        __u32 xres_virtual;             /* virtual resolution           */
        __u32 yres_virtual;
        __u32 xoffset;                  /* offset from virtual to visible */
        __u32 yoffset;                  /* resolution                   */

        __u32 bits_per_pixel;           /* guess what                   */
        __u32 grayscale;                /* 0 = color, 1 = grayscale,    */
                                        /* >1 = FOURCC                  */
        struct fb_bitfield red;         /* bitfield in fb mem if true color, */
        struct fb_bitfield green;       /* else only length is significant */
        struct fb_bitfield blue;
        struct fb_bitfield transp;      /* transparency                 */

        __u32 nonstd;                   /* != 0 Non standard pixel format */

        __u32 activate;                 /* see FB_ACTIVATE_*            */

        __u32 height;                   /* height of picture in mm    */
        __u32 width;                    /* width of picture in mm     */

        __u32 accel_flags;              /* (OBSOLETE) see fb_info.flags */

        /* Timing: All values in pixclocks, except pixclock (of course) */
        __u32 pixclock;                 /* pixel clock in ps (pico seconds) */
        __u32 left_margin;              /* time from sync to picture    */
        __u32 right_margin;             /* time from picture to sync    */
        __u32 upper_margin;             /* time from sync to picture    */
        __u32 lower_margin;
        __u32 hsync_len;                /* length of horizontal sync    */
        __u32 vsync_len;                /* length of vertical sync      */
        __u32 sync;                     /* see FB_SYNC_*                */
        __u32 vmode;                    /* see FB_VMODE_*               */
        __u32 rotate;                   /* angle we rotate counter clockwise */
        __u32 colorspace;               /* colorspace for FOURCC-based modes */
        __u32 reserved[4];              /* Reserved for future compatibility */
};
```

Now that we know all about these structures, we can use ioctl on our open file descriptor to fill these structures.

```
        //Get variable screen information
        ioctl(fb_fd, FBIOGET_VSCREENINFO, &vinfo);

        //Get fixed screen information
        ioctl(fb_fd, FBIOGET_FSCREENINFO, &finfo);
```

Note that the fb_var_screeninfo structure is *variable* information. This means, in addition to the FBIOGET_VSCREENINFO, we can also call ioctl with FBIOPUT_VSCREENINFO to *change* the settings of the framebuffer. Most importantly, we probably will want to set the bits_per_pixel field to something reasonable, since by default it seems to be set to something like 8, and is not enough to render in color. You might also need to set grayscale to 0, but in practice, it seems to work even if you don't do that. After that, you should get it again to make sure that your changes were successful.

```
        //Get variable screen information
        ioctl(fb_fd, FBIOGET_VSCREENINFO, &vinfo);
        vinfo.grayscale=0;
        vinfo.bits_per_pixel=32;
        ioctl(fb_fd, FBIOPUT_VSCREENINFO, &vinfo);
        ioctl(fb_fd, FBIOGET_VSCREENINFO, &vinfo);
```

Once that is done, we can calculate the total size of the screen (in bytes). This is important because we will need to map exactly the right amount of memory, and only draw into that memory, otherwise bad things will happen. To calculate the size of the screen (the size of the buffer), we can use vinfo.yres_virtual, which is the number of horizontal lines on the screen, multiplied by finfo.line_length, the length of each line in bytes.

```
        long screensize = vinfo.yres_virtual * finfo.line_length;
```

Once we have the size of the screen, we can use mmap to map the buffer to memory. mmap will return a pointer to the beginning of the memory.

```
        uint8_t *fbp = mmap(0, screensize, PROT_READ | PROT_WRITE, MAP_SHARED, fb_fd,
(off_t)0);
```

Now, you have your framebuffer mapped to memory. All that is left to do is to draw on it. This we can do just by setting the memory at the right location to the correct value of the pixel in the color you want. So next what we need to do is calculate the correct location in the mapped memory of the pixel that we want to set. For this, we can use the following algorithm:

```
        long x,y; //location we want to draw the pixel
        uint32_t pixel; //The pixel we want to draw at that location

        //Make sure you set x,y and pixel correctly (details later)

        long location = (x+vinfo.xoffset) * (vinfo.bits_per_pixel/8) + (y+vinfo.yoffset) *
finfo.line_length;
        *((uint32_t*)(fbp + location)) = pixel;
```

`(y+vinfo.yoffset) * finfo.line_length` gets the beginning of line y in memory, and `(x+vinfo.xoffset) * (vinfo.bits_per_pixel/8)` gets the offset of x on that line. All we have to do is add those together and we have the correct location in memory of the pixel we want to draw. Then we just set that memory to the pixel we want to draw. So we need to decide what color we want the pixel to be and then calculate what the value for a pixel of that color would be. We can use the vinfo structure to figure out the pixel format required for computing the correct pixel value, specifically the red, green, and blue fields, which are also structures. I like to write a little function that takes 8 bit values for each color and returns the pixel value, it looks like this:

```
inline uint32_t pixel_color(uint8_t r, uint8_t g, uint8_t b, struct fb_var_screeninfo *vinfo)
{
        return (r<<vinfo->red.offset) | (g<<vinfo->green.offset) | (b<<vinfo->blue.offset);
}
```

This function takes the 8 bit value and shifts it to the left the correct offset of that color. Then combines it with the other colors using the OR operator. So if we want to draw a pixel of color 0xFF,0x00,0xFF (purple), it take the red value (0xFF), shift it over the correct offset or red (probably 16) and the result would be 0x00FF0000, then it would take the green value (0x00) and shift that to the left (probably 8 bits) and then OR those together, resulting in the same value since green was set to 0, and then take blue (0xFF) and shift that the left (probably 0 bits) resulting in 0x000000FF, then OR that value with the red and green to get the final pixel color of 0x00FF00FF.

So you can see how easy it is to determine the correct pixel value. Now one important thing you need to remember and be aware of at all time is that you must never try to draw outside of the screen. This is because your program has gotten permission to write into that buffer, but if you try to write outside that buffer, you are essentially trying to modify somebody else's memory. Linux probably won't allow this and your program will end with a segfault. This is bad for your program, but if you have set the tty to graphics mode (I will explain later), then you can cause the whole machine to lock up, which is a very bad thing to do. To make sure we don't draw outside the screen, we can use `vinfo.xres` (the width of the screen in pixels) and `vinfo.yres` (the height of the screen in pixels). If you never draw and pixels above vinfo.xres,vinfo.yres or below 0,0, then you should be fine. Also, you should note that it is actually safe to draw X values past vinfo.xres since the buffer is just one big block of memory, if you exceed vinfo.xres, you will actually be drawing on the line below Y. So (1+vinfo.xres),14 is the same as 1,15. Of course, this is probably not something that you would ever want to do, so it's probably best to just never draw X greater than vinfo.xres.

Now, as a final example, let's take a look at some code to set up the framebuffer and then make the entire screen bright purple. The code looks like this:

```
#include <linux/fb.h>
#include <stdio.h>
#include <stdint.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/ioctl.h>

inline uint32_t pixel_color(uint8_t r, uint8_t g, uint8_t b, struct fb_var_screeninfo *vinfo)
{
        return (r<<vinfo->red.offset) | (g<<vinfo->green.offset) | (b<<vinfo->blue.offset);
}

int main()
{
        struct fb_fix_screeninfo finfo;
        struct fb_var_screeninfo vinfo;
        int fb_fd = open("/dev/fb0",O_RDWR);

        //Get variable screen information
        ioctl(fb_fd, FBIOGET_VSCREENINFO, &vinfo);
        vinfo.grayscale=0;
        vinfo.bits_per_pixel=32;
        ioctl(fb_fd, FBIOPUT_VSCREENINFO, &vinfo);
        ioctl(fb_fd, FBIOGET_VSCREENINFO, &vinfo);

        ioctl(fb_fd, FBIOGET_FSCREENINFO, &finfo);

        long screensize = vinfo.yres_virtual * finfo.line_length;

        uint8_t *fbp = mmap(0, screensize, PROT_READ | PROT_WRITE, MAP_SHARED, fb_fd,
(off_t)0);

        int x,y;

        for (x=0;x<vinfo.xres;x++)
                for (y=0;y<vinfo.yres;y++)
                {
                        long location = (x+vinfo.xoffset) * (vinfo.bits_per_pixel/8) +
(y+vinfo.yoffset) * finfo.line_length;
                        *((uint32_t*)(fbp + location)) = pixel_color(0xFF,0x00,0xFF, &vinfo);
                }

        return 0;
}
```

Now if you run this code, you will probably see nothing, that doesn't mean that it's not working, it just means your eyes aren't fast enough. If you want to see it work, you should add a delay (nanosleep) after rendering.

I have one final thing to explain, as I promised earlier. However, you need to be aware that this is dangerous and not strictly required. If you use this and something goes wrong and your program doesn't clean up properly, then you will lock up your computer (not really, but the screen will stop responding). You have been warned.

What I am talking about is claiming the tty for graphics only. This will prevent the framebuffer console from trying to draw overtop of your graphics. You will need to use ioctl again, but this time not on the framebuffer device, instead you will have to use it on the tty device, probably /dev/tty0. You will need to call KDSETMODE with KD_GRAPHICS to set up the tty for graphics only, and then with KD_TEXT to undo it. You *MUST* set it back to KD_TEXT at exit, or else. I recommend never using this until you are 100% sure that your code will not cause a segfault, otherwise you are going to be rebooting your computer a lot. You could probably also set something up so that you can press some key combination or type some simple command that runs a program that just sets KDSETMODE back to KD_TEXT, that's probably easier, but somehow I doubt you will actually bother. Anyways, the code looks like this:

```
        int tty_fd = open("/dev/tty0", O_RDWR);
        ioctl(tty_fd,KDSETMODE,KD_GRAPHICS);


        ...


        //At exit:
        ioctl(tty_fd,KDSETMODE,KD_TEXT);
```

That's all there is to it.

Now that you are all set up to render stuff, you probably want to start actually drawing stuff besides single pixels or clearing the whole screen, and you probably want to to do something about the flickering that will occur when rendering directly to the framebuffer. If that's the case, have a look at the sections on double buffering and drawing primatives.

Direct Rendering Manager (DRM) Dumb Buffers

Supposedly, fbdev is the "old" way of doing things, and KMS/DRM is the "new" way. I don't really get what all the fuss is about, I like fbdev for what it is. KMS/DRM has certain obvious advantages, but that's no reason to throw away fbdev (fortunately, fbdev will still be around for a long time though).

Anyways, KMS/DRM is a much more featureful interface, and gives you a lot more options (which means it's also a lot more complicated). KMS/DRM offers much more control over the graphics hardware, which is great if you want to do really fancy stuff like hardware acceleration. It also has it's own mechanism for double buffering, which is nice.

Anyways, let's get to it. DRM has a feature called "dumb buffers" which is essentially a framebuffer. It's supposedly the easiest to set up, but really still quite a pain. KMS/DRM is a kernel interface, however, most applications using KMS/DRM use libdrm, which makes some parts of the process a lot easier. However, this is a low-level tutorial, and in order to keep this as low-level as possible, we are going to avoid using any user land library. Fortunately, it isn't too difficult to bypass libdrm (thankfully everything is open source).

First of all, we need to discover and configure the hardware. For this, we will use Kernel Mode Setting (KMS). This step is quite tedious, but bare with me. A lot of this typically handled in libdrm, but we will not be using libdrm at all. I learned a lot of this by reading the libdrm source code, I have simplified a lot of it.

The very first thing that needs to be done is to open the DRI device (DRI stands for Direct Rendering Infrastructure). The DRI device is provided by the kernel in /dev/dri/card0. This assumes you only have one graphics card of course, if you have more than one graphics card, they will be called card1, card2, card3, etc.... However, it is usually safe to assume that you can just use card0. We can open it with a call to open, just like we did for fbdev.

```
        int dri_fd  = open("/dev/dri/card0",O_RDWR);
```

We will need this file descriptor to do all our communication with the DRM driver through ioctl calls. We will also need to have some structures and preprocessor constants, all of which are defined in two header files provided by the kernel. Include the headers drm/drm.h and drm/drm_mode.h. Do not include xf86drm.h or xf86drmMode.h, these are part of libdrm, not the kernel interface.

```
#include <drm/drm.h>
#include <drm/drm_mode.h>
```

The first thing we need to do is become the "master" of the dri device. This we can do with the ioctl, DRM_IOCTL_SET_MASTER.

```
        ioctl(dri_fd, DRM_IOCTL_SET_MASTER, 0);
```

We only need to be "master" to do the actual KMS mode setting, so we can drop it as soon as we are done with that.

The first thing we need to do it find all the "connectors" the card provides. "Connectors" seem to usually correspond to actual hardware connectors on the graphics card. For instance, I have 3 on my laptop: The internal connector to the LCD, the external VGA connector and the HDMI connector.

To get this we need to create an instance of the structure drm_mode_card_res. We can ask the kernel to fill this structure with the ioctl, DRM_IOCTL_MODE_GETRESOURCES. Let's quick take a look at the structure's definition

```
struct drm_mode_card_res {
        __u64 fb_id_ptr;
        __u64 crtc_id_ptr;
        __u64 connector_id_ptr;
        __u64 encoder_id_ptr;
        __u32 count_fbs;
        __u32 count_crtcs;
        __u32 count_connectors;
        __u32 count_encoders;
        __u32 min_width, max_width;
        __u32 min_height, max_height;
};
```

However, the way that we have to do this is a little funky because we don't actually know how many connectors there are yet. You may notice that the structure contains a field called count_connectors, which will be used to notify our program of how many connectors there are on the card. Then, once we have this count, we will call ioctl again and get the actual connector ID, which we can then use to get the actual connector information. First thing's first, we need to make sure that the structure is zero'ed out, otherwise the kernel will interpret the ioctl as a request for IDs instead of a request for the resource counts. Then we use DRM_IOCTL_MODE_GETRESOURCES to fill in the count fields of the structure.

This is where there might be trouble, depending on your application. You are supposed to allocate enough memory to store the IDs of the resources. I would rather not allocate memory dynamically if I don't have to, although you may feel free to do so if it suits your purposes. Instead, I am just going to allocate more than enough space and assume that it will always be enough. This might be considered bad form to some people, as it makes assumptions about the user's hardware and wastes a tiny bit of memory, but I am happy with it. We should allocate this memory as an array of 64 bit integers, and we will need four of them since we have four types of resources to deal with (connectors, encoders, framebuffers, and crtcs). The code will look something like this.

```
        uint64_t res_fb_buf[10]={0},
                        res_crtc_buf[10]={0},
                        res_conn_buf[10]={0},
                        res_enc_buf[10]={0};

        struct drm_mode_card_res res={0};

        //Get resource counts
        ioctl(dri_fd, DRM_IOCTL_MODE_GETRESOURCES, &res);

        res.fb_id_ptr=(uint64_t)res_fb_buf;

        res.crtc_id_ptr=(uint64_t)res_crtc_buf;

        res.connector_id_ptr=(uint64_t)res_conn_buf;

        res.encoder_id_ptr=(uint64_t)res_enc_buf;

        //Get resource IDs

        ioctl(dri_fd, DRM_IOCTL_MODE_GETRESOURCES, &res);
```

Note how we filled in the fields fb_id_ptr, crtc_id_ptr, connector_id_ptr, and encoder_id_ptr with the respective memory addresses of the arrays we created.

Then, the next thing we need to do is iterate through all the connector IDs we discovered and get the information about them. Once we get enough information, we can actually do the mode setting to set the card up. Fortunately, the kernel driver has already provided us with the number of connectors on the graphics card, so we can set up our loop very easily.

```
        int i;

        for (i=0;i<res.count_connectors;i++)

        {
```

Then, we can start to get information about the connector. For this we will need a structure called drm_mode_get_connector, which is defined like this.

```
struct drm_mode_get_connector {

        __u64 encoders_ptr;

        __u64 modes_ptr;

        __u64 props_ptr;

        __u64 prop_values_ptr;


        __u32 count_modes;

        __u32 count_props;

        __u32 count_encoders;


        __u32 encoder_id; /** Current Encoder */

        __u32 connector_id; /** Id */

        __u32 connector_type;

        __u32 connector_type_id;
        __u32 connection;
        __u32 mm_width, mm_height; /** HxW in millimeters */
        __u32 subpixel;
};
```

We will need to use ioctl DRM_IOCTL_MODE_GETCONNECTOR to fill in this structure. Like DRM_IOCTL_MODE_GETRESOURCES, we will need to use this ioctl twice, first to get the resource counts, then next to get the resources.

It's important to know about the connector's resources, especially what it calls "modes". "Modes", in this case, are the available resolutions for the display you are using, starting with the current one. This will tell us how big we can make our framebuffer. Notice here that the fields resource fields do not include _id_ptr, but just _ptr. This time, the kernel will not fill in the allocated memory with resource IDs, but with actual resource information, so we need to actually create an array of structures for the modes_ptr field. This structure should be of type drm_mode_modeinfo which is defined as follows.

```
struct drm_mode_modeinfo
{
        __u32 clock;
        __u16 hdisplay, hsync_start, hsync_end, htotal, hskew;
        __u16 vdisplay, vsync_start, vsync_end, vtotal, vscan;

        __u32 vrefresh;

        __u32 flags;
        __u32 type;
        char name[DRM_DISPLAY_MODE_LEN];
};
```

The other resource fields can just be unsigned 64 bit integers, we probably won't even need to look at them, but they do need to exist so that we don't overwrite some memory somewhere else by accident. We can perform that ioctl now.

```
                struct drm_mode_modeinfo conn_mode_buf[20]={0};
                uint64_t        conn_prop_buf[20]={0},
                                        conn_propval_buf[20]={0},
                                        conn_enc_buf[20]={0};
                struct drm_mode_get_connector conn={0};

                conn.connector_id=res_conn_buf[i];

                ioctl(dri_fd, DRM_IOCTL_MODE_GETCONNECTOR, &conn);      //get connector
resource counts
                conn.modes_ptr=(uint64_t)conn_mode_buf;
                conn.props_ptr=(uint64_t)conn_prop_buf;
                conn.prop_values_ptr=(uint64_t)conn_propval_buf;
                conn.encoders_ptr=(uint64_t)conn_enc_buf;
                ioctl(dri_fd, DRM_IOCTL_MODE_GETCONNECTOR, &conn);      //get connector
resource IDs
```

Now that we have our "modes", we can create our "dumb" framebuffer. We don't want to try to create any framebuffers for non-connected connectors though, so we can can do a few checks to see if it's valid. First, we can check if there is at least 1 valid encoder, and one valid mode. We will need both a valid mode and a valid encoder in order to make use of the connector, so if there aren't any, then there is no need to mess with that connector further. In addition, we can check if the connector is "connected", with the connection field. We can also check if the connector has an active encoder. We could still use it even if it has no active encoder, but it's a lot of work, and one connector will have an active encoder. So we can check if there is an encoder_id.

```
                if (conn.count_encoders<1 || conn.count_modes<1 || !conn.encoder_id ||
!conn.connection)
                        continue;
```

This part is nice an easy compared with KMS, except for one small compilation, which is what to do in the case that there is more than one connector active. One thing we can do is create a framebuffer for each connector, but this could lead to complications later when we want to draw to them. We might also just assume that there is only one valid connector, but this is pretty ugly. It is also possible to use KMS to set up the system with one framebuffer for all the valid connectors. What you do here is really dependent on what your program wants to do. For this example, we will create a framebuffer for each connector.

So what we have to do is maintain a list of all the created framebuffers and the mode that you want to set it to. I am going to do this with just three arrays, all the same size as our maximum assumed number of connectors. The first will store the base pointer for the framebuffer, and the other two will hold the width and height for the buffers. It's an ugly solution, but this will work.

```
        void *fb_base[10];
        long fb_w[10];
        long fb_h[10];
```

SIDE NOTE: You may notice that our code is becoming uglier and uglier. The reason for this is because we are avoiding dealing with some fairly serious issues. Namely how we should actually choose a mode, how to choose a connector, what to do with more than one connector, and also avoiding dynamic memory allocation whenever possible. There are elegant solutions to these problems, but outside the scope of this document. However, the basic outline of the elegant solution is as follows, check each connector for validity, build a list of all valid connectors, find a mode valid for all valid connectors, create a single framebuffer at this resolution, set up all connectors to use this framebuffer. END SIDE NOTE

Next, let's create our "dumb" framebuffer. We need three structures, drm_mode_create_dumb, drm_mode_map_dumb, and drm_mode_fb_cmd.

```
                struct drm_mode_create_dumb create_dumb={0};
                struct drm_mode_map_dumb map_dumb={0};
                struct drm_mode_fb_cmd cmd_dumb={0};
```

The first two are used to create and map the dumb buffer. The last one is used to "add" the buffer, basically letting the DRM driver know that we created it. The structures are defined as follows.

```
struct drm_mode_create_dumb {
        __u32 height;
        __u32 width;
        __u32 bpp;
        __u32 flags;

        __u32 handle;
        __u32 pitch;
        __u64 size;
};
struct drm_mode_fb_cmd {
        __u32 fb_id;
        __u32 width, height;
        __u32 pitch;
        __u32 bpp;
        __u32 depth;
        /* driver specific handle */
        __u32 handle;
};
struct drm_mode_map_dumb {
        __u32 handle;
        __u32 pad;

        __u64 offset;
};
```

The first thing we need to do is fill out the create structure with a mode. Then we can call the ioctl, DRM_IOCTL_MODE_CREATE_DUMB. This will fill in the handle field. This handle is a Graphics Execution Manager (GEM) handle, and we will need to remember it. Second, we put the handle into the drm_mode_fb_cmd structure (as well as other fields) and call the ioctl, DRM_IOCTL_MODE_ADDFB. Then we prepare the framebuffer for mapping by filling in the drm_mode_map_dumb structure's handle and using the ioctl, DRM_IOCTL_MODE_MAP_DUMB. Then we can finally map the framebuffer using mmap.

```
                create_dumb.width = conn_mode_buf[0].hdisplay;
                create_dumb.height = conn_mode_buf[0].vdisplay;
                create_dumb.bpp = 32;
                create_dumb.flags = 0;
                create_dumb.pitch = 0;
                create_dumb.size = 0;
                create_dumb.handle = 0;
                ioctl(dri_fd, DRM_IOCTL_MODE_CREATE_DUMB, &create_dumb);

                cmd_dumb.width=create_dumb.width;
                cmd_dumb.height=create_dumb.height;
                cmd_dumb.bpp=create_dumb.bpp;
                cmd_dumb.pitch=create_dumb.pitch;
                cmd_dumb.depth=24;
                cmd_dumb.handle=create_dumb.handle;
                ioctl(dri_fd,DRM_IOCTL_MODE_ADDFB,&cmd_dumb);
                map_dumb.handle=create_dumb.handle;
                ioctl(dri_fd,DRM_IOCTL_MODE_MAP_DUMB,&map_dumb);

                fb_base[i] = mmap(0, create_dumb.size, PROT_READ | PROT_WRITE, MAP_SHARED,
dri_fd, map_dumb.offset);
                fb_w[i]=create_dumb.width;
                fb_h[i]=create_dumb.height;
```

Note how I also filled out the arrays for the various connectors with the base pointer to the mapped framebuffer and the corresponding dimension of them.

Then we can be done with the dumb buffer creation. We now have our framebuffer(s) mapped and once we finish our mode setting we can begin drawing into them.

Back to mode setting. Next we need to get an encoder. In this context, an encoder is what takes our framebuffer and gets it into the proper format for "scanout" (that is, displaying it on the screen). Fortunately, our connector structure has a great field called encoder_id, which is the ID of the active encoder for that connector. We could search for other encoders, but we could also just use this one, let's do that. We only have the ID of the encoder, but we will need to get more information about it. For this, use the drm_mode_get_encoder structure along with the ioctl, DRM_IOCTL_MODE_GETENCODER. The structure looks like this.

```
struct drm_mode_get_encoder {
        __u32 encoder_id;
        __u32 encoder_type;

        __u32 crtc_id; /** Id of crtc */

        __u32 possible_crtcs;
        __u32 possible_clones;
};
```

Which gives us the ID of the current CRTC for this encoder. A CRTC is a CRT (Cathode Ray Tube??) Controller. This is what we really need to set up. The CRTC's structure looks like this.

```
struct drm_mode_crtc {
        __u64 set_connectors_ptr;
        __u32 count_connectors;

        __u32 crtc_id; /** Id */
        __u32 fb_id; /** Id of framebuffer */


        __u32 x, y; /** Position on the frameuffer */


        __u32 gamma_size;
        __u32 mode_valid;
        struct drm_mode_modeinfo mode;
};
```

We can get information about it using the ioctl, DRM_IOCTL_MODE_GETCRTC. Our code might look like this.

```
                struct drm_mode_get_encoder enc={0};


                enc.encoder_id=conn.encoder_id;
                ioctl(dri_fd, DRM_IOCTL_MODE_GETENCODER, &enc); //get encoder


                struct drm_mode_crtc crtc={0};


                crtc.crtc_id=enc.crtc_id;
                ioctl(dri_fd, DRM_IOCTL_MODE_GETCRTC, &crtc);
```

Then, finally, we can set up the CRTC and connect it to our newly created dumb framebuffer using the ioctl, DRM_IOCTL_MODE_SETCRTC. Take note that mode_valid must be set to 1, otherwise the DRM driver won't do anything.

```
                crtc.fb_id=cmd_dumb.fb_id;
                crtc.set_connectors_ptr=(uint64_t)&res_conn_buf[i];
                crtc.count_connectors=1;
                crtc.mode=conn_mode_buf[0];
                crtc.mode_valid=1;
                ioctl(dri_fd, DRM_IOCTL_MODE_SETCRTC, &crtc);
        }
```

I set up the connectors_ptr field with sort of a hack here. I just used the original list, but instead of starting at the beginning, I start at the current position in the list (the current connector) and hard coded the connector count to always be one. At this point you can stop being the "master" of the DRM device as this is no longer required. Use the ioctl, DRM_IOCTL_DROP_MASTER.

```
        ioctl(dri_fd, DRM_IOCTL_DROP_MASTER, 0);
```

Then we are finally done with our KMS. We can now draw on our framebuffer and the DRM driver will deliver that data to the screen. For instance, to clear the screen to the color purple, we might use the following code.

```
        int i;

        for (i=0;i<res.count_connectors;i++)

                for (y=0;y<fb_h[i];y++)

                        for (x=0;x<fb_w[i];x++)

                        {

                                int location=y*(fb_w[i]) + x;

                                *(((uint32_t*)fb_base[i])+location)=0x00ff00ff;

                        }
```

That's all there is to it. A full working example follows.

```
#include <stdio.h>

#include <stdint.h<

#include <fcntl.h<

#include <sys/mman.h<

#include <sys/ioctl.h<

#include <drm/drm.h<

#include <drm/drm_mode.h<


int main()

{

//------------------------------------------------------------------------------

//Opening the DRI device

//------------------------------------------------------------------------------


        int dri_fd  = open("/dev/dri/card0",O_RDWR | O_CLOEXEC);

```

```
//------------------------------------------------------------------------------
//Kernel Mode Setting (KMS)
//------------------------------------------------------------------------------

        uint64_t res_fb_buf[10]={0},
                        res_crtc_buf[10]={0},
                        res_conn_buf[10]={0},
                        res_enc_buf[10]={0};

        struct drm_mode_card_res res={0};

        //Become the "master" of the DRI device
        ioctl(dri_fd, DRM_IOCTL_SET_MASTER, 0);

        //Get resource counts
        ioctl(dri_fd, DRM_IOCTL_MODE_GETRESOURCES, &res);
        res.fb_id_ptr=(uint64_t)res_fb_buf;
        res.crtc_id_ptr=(uint64_t)res_crtc_buf;
        res.connector_id_ptr=(uint64_t)res_conn_buf;
        res.encoder_id_ptr=(uint64_t)res_enc_buf;
        //Get resource IDs
        ioctl(dri_fd, DRM_IOCTL_MODE_GETRESOURCES, &res);

        printf("fb: %d, crtc: %d, conn: %d, enc:
%d\n",res.count_fbs,res.count_crtcs,res.count_connectors,res.count_encoders);

        void *fb_base[10];
        long fb_w[10];
        long fb_h[10];

        //Loop though all available connectors
        int i;
        for (i=0;i<res.count_connectors;i++)
        {
                struct drm_mode_modeinfo conn_mode_buf[20]={0};
                uint64_t        conn_prop_buf[20]={0},
                                        conn_propval_buf[20]={0},
                                        conn_enc_buf[20]={0};

                struct drm_mode_get_connector conn={0};

                conn.connector_id=res_conn_buf[i];

                ioctl(dri_fd, DRM_IOCTL_MODE_GETCONNECTOR, &conn);      //get connector
resource counts
                conn.modes_ptr=(uint64_t)conn_mode_buf;
                conn.props_ptr=(uint64_t)conn_prop_buf;
                conn.prop_values_ptr=(uint64_t)conn_propval_buf;
                conn.encoders_ptr=(uint64_t)conn_enc_buf;
                ioctl(dri_fd, DRM_IOCTL_MODE_GETCONNECTOR, &conn);      //get connector
resources

                //Check if the connector is OK to use (connected to something)
                if (conn.count_encoders<1 || conn.count_modes<1 || !conn.encoder_id ||
!conn.connection)
                {
                        printf("Not connected\n");
                        continue;
                }

//------------------------------------------------------------------------------
//Creating a dumb buffer
//------------------------------------------------------------------------------
                struct drm_mode_create_dumb create_dumb={0};
                struct drm_mode_map_dumb map_dumb={0};
                struct drm_mode_fb_cmd cmd_dumb={0};

                //If we create the buffer later, we can get the size of the screen first.
                //This must be a valid mode, so it's probably best to do this after we find
                //a valid crtc with modes.
                create_dumb.width = conn_mode_buf[0].hdisplay;
                create_dumb.height = conn_mode_buf[0].vdisplay;
                create_dumb.bpp = 32;
                create_dumb.flags = 0;
                create_dumb.pitch = 0;
                create_dumb.size = 0;
                create_dumb.handle = 0;
                ioctl(dri_fd, DRM_IOCTL_MODE_CREATE_DUMB, &create_dumb);

                cmd_dumb.width=create_dumb.width;
                cmd_dumb.height=create_dumb.height;
                cmd_dumb.bpp=create_dumb.bpp;
                cmd_dumb.pitch=create_dumb.pitch;
                cmd_dumb.depth=24;
                cmd_dumb.handle=create_dumb.handle;
                ioctl(dri_fd,DRM_IOCTL_MODE_ADDFB,&cmd_dumb);

                map_dumb.handle=create_dumb.handle;
                ioctl(dri_fd,DRM_IOCTL_MODE_MAP_DUMB,&map_dumb);

                fb_base[i] = mmap(0, create_dumb.size, PROT_READ | PROT_WRITE, MAP_SHARED,
dri_fd, map_dumb.offset);
                fb_w[i]=create_dumb.width;
                fb_h[i]=create_dumb.height;

//------------------------------------------------------------------------------
//Kernel Mode Setting (KMS)
//------------------------------------------------------------------------------

                printf("%d : mode: %d, prop: %d, enc:
%d\n",conn.connection,conn.count_modes,conn.count_props,conn.count_encoders);
                printf("modes: %dx%d FB:
%d\n",conn_mode_buf[0].hdisplay,conn_mode_buf[0].vdisplay,fb_base[i]);

                struct drm_mode_get_encoder enc={0};

                enc.encoder_id=conn.encoder_id;
                ioctl(dri_fd, DRM_IOCTL_MODE_GETENCODER, &enc); //get encoder
                struct drm_mode_crtc crtc={0};

                crtc.crtc_id=enc.crtc_id;
                ioctl(dri_fd, DRM_IOCTL_MODE_GETCRTC, &crtc);

                crtc.fb_id=cmd_dumb.fb_id;
                crtc.set_connectors_ptr=(uint64_t)&res_conn_buf[i];
                crtc.count_connectors=1;
                crtc.mode=conn_mode_buf[0];
                crtc.mode_valid=1;
                ioctl(dri_fd, DRM_IOCTL_MODE_SETCRTC, &crtc);
        }

        //Stop being the "master" of the DRI device
        ioctl(dri_fd, DRM_IOCTL_DROP_MASTER, 0);

        int x,y;
        for (i=0;i<100;i++)
        {
                int j;
                for (j=0;j<res.count_connectors;j++)
                {
                        int col=(rand()%0x00ffffff)&0x00ff00ff;
                        for (y=0;y<fb_h[j];y++)
                                for (x=0;x<fb_w[j];x++)
                                {
                                        int location=y*(fb_w[j]) + x;
                                        *(((uint32_t*)fb_base[j])+location)=col;
                                }
                }
                usleep(100000);
        }

        return 0;
}
```

Now you should be able to get graphics drawn on the screen using KMS/DRM dumb buffers. For further information, you should check out the sections on double buffering and drawing primatives.

PLEASE NOTE: This section provides a working example, but it still needs a significant amount of work to improve it. It shows many less than beautiful solutions for the problems faced. If you have a better solution, please email me at prushik@betteros.org and I will update the tutorial (and include your name). However, solutions must follow the following constraints: no libdrm, no standard libc functions that do not map 1-1 to Linux system calls, no 3rd party libraries.

X Server (X11) Direct Connection

Connecting to the X11 server is by far the most common method of displaying graphics on Linux systems. It has been around a long time and is in use on virtually every Linux system, it will most likely still be around for the forseeable future, even after Wayland and Mir start to see common use. However, X11 is far from a simple and easy to use protocol. Typically, an application using X11 for graphics will use a very high level widget library, such as GTK+ or QT, which in turn use Xlib or XCB to establish connection and handle communication with the X server. A simpler application might only use Xlib or XCB if the programmer has enough skill. XCB is currently accepted as the lowest level method possible of communicating with the X server. However, I refuse to accept this.

The X11 protocol uses the client server model for communication. This means that, if we can open sockets, we can connect to the X server on our own, without relying on Xlib or XCB to facilitate communication. We will just have to handle the X11 protocol stuff ourselves. If you were writing an X server, this would be a very daunting and nearly impossible task given the scope of all the X server is expected to handle; however, writing a client is a much simpler task as you will only need to implement the parts of the protocol you need to work with, and you can ignore all the extensions you aren't interested in.

However, it's important to note that Xorg actually allows for multiple transport methods. The most common of which is actually not network communication, but it is socket based communication, so the connection gets established in almost the same way.

The first thing you will need to do in order to establish a connection is to check out the DISPLAY environment variable. The DISPLAY environment variable will give us the information about which transport method to use and where the server is located. This variable will only be set on the client, which is what we are focusing on here. The DISPLAY variable has three parts, which you will have to be able to parse out. The DISPLAY variable come in the following form:

`host:display.screen` *host* is the hostname of the server. This field is is actually the one that determines the transport method. If a host name is specified and a valid and resolvable hostname or an IP address, then

TCP is selected as the transport method. If this field is left blank, or is "unix", then Unix Domain Sockets are used. On modern desktop systems, DISPLAY is most likely going to be set to ":0.0", so supporting Unix Domain Sockets is probably going to be very important.
display and *screen* are the display number and screen number. Screen is an optional field, if it is left out, the application should assume 0. Multiple screens are hardly ever used, even on setups which actually do have more than one monitor attached, so ignoring screens might be good enough, it's up to you. display is manditory, but it is almost always zero, in fact, it's so common that some video card drivers actually don't work right if it the display number is anything else. However, it's always best to do handle every scenario as long as performance isn't impeded. If you are unsure of what any of these terms mean, then you should do some research about X11 and how it works.

So now you should be able to see how we can find out the server to connect to and the transport method using only the DISPLAY variable. Now that we know this, we can establish our connection.

In the case that our server is using TCP as a transport method, then the address is already in the DISPLAY variable, the TCP port is a little trickier to figure out though. The typical X11 port is TCP port 6000, however, if the display set in the DISPLAY variable is greater than zero, the port number the server will be listening on will be 6000 + display number. So if our DISPLAY variable is set to "127.0.0.1:14.0", then we would want to connect to the localhost (127.0.0.1 is localhost) on port 6014.

In the more likely case that the server is using Unix Domain Sockets, then we don't have to worry about the address or hostname to connect to, since Unix Domain Sockets are only for inter process communication on one machine. If you aren't familiar with Unix Domain Sockets, I'll explain a little bit about them. It is a very similar concept to a Linux kernel device file. The server adds an entry to the file hierachy, which appears as a file. The client can then connect to that node using it's path and then communicate with the server by writing into the resulting file descriptor. In the case of X11, the server creates a node in /tmp/.X11-unix/ named after the display number prefixed by a capital X. It seems scary to me to have to hard-code paths into my application, but I checked the XCB sources, and it seems to be exactly what XCB is doing anyway. So therefore, if DISPLAY is ":14.0", then our client application would need to connect to /tmp/.X11-unix/X14. We don't have to worry about port numbers when using Unix Domain Sockets, since there aren't any in this transport method.
Here is an example of how to connect to a Unix Domain Socket for display ":0"

```
        //Create the socket
        sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
        if (sockfd < 0)
                error("Error opening socket\n",21);

        serv_addr.sun_family = AF_UNIX;
        strcpy(serv_addr.sun_path, "/tmp/.X11-unix/X0");
        srv_len = sizeof(struct sockaddr_un);


        //Connect to socket
        connect(sockfd,(struct sockaddr *)&serv_addr,srv_len);
```

Once the connection is established, we can start communicating directly with the server using normal read/write calls, all we have to do is get the protocol right. If you have ever programmed anything with XCB, then you should already have a good idea of what needs to be done, since XCB functions are almost a 1 to 1 mapping of packets sent to the server. One good way to get an idea of what your packets should look like is to do a network capture of a simple X11 or XCB program. To do this, make sure your X server supports TCP communication (wasn't run with -nolisten flag), then set your DISPLAY variable to "127.0.0.1:0", and then run your packet capture software and the test program.

Basically, we need to do the same things that you would normally do in XCB. However, before we get into that, we have to finish setting up the connection. The X11 protocol starts with a little handshake, the client send an initial connection request, to which the server responds with an initial connection reply. The request is a short and simple packet, which basically only contains 2 pieces of information, the byte order the connection will use, and the protocol version that will be used.

The byte order field is first and it is just a one byte wide field. This should probably be set to little endian unless you are working with embedded systems. For little endian, the field should be set to 0x6c which is a lower case 'l'. For big endian, this should be 0x66 which is an upper case 'B'. Immediately following the endianness field, is one byte of padding. This can probably be anything, but set it to zero just in case.

Next are two 16 bit wide integer fields specifying the protocol version, the major version number followed by the minor version number. Since we are using X11, the major version should just be 11 (0x0b). The minor number should be 0. This is what all clients I have seen do, and I doubt there is any reason to set the minor version to anything different.

Following that are two more 16 bit wide fields. These are used for an authorization protocol, which I don't want to get into or deal with, so just set them both to 0. I have never seen anything else anyway. Finally, following that, there is another two bytes of unused padding.

The request packets fields can be easily stored in a C structure like the following:

```
struct x11_conn_req
{
        uint8_t order;
        uint8_t pad1;
        uint16_t major,
                       minor;
        uint16_t auth_proto,
                        auth_data;
        uint16_t pad2;
};
```

That concludes the connection request part. Assuming that the server accepts your connection, it will finish the handshake with a connection reply. This reply is generally a massive packet containing a ton of information, which will not be fun to decode, but it has to be done.

The first thing you need to do is read the first 8 bytes. The packet will be bigger than that, but we don't want to read the rest of it yet because it may vary in size, and we don't know yet how big it will be. The first 8 bytes will contain the information that we need to know to get the rest of the packet.

The reply will contain a one byte field called a success code, then one byte of padding, then two 16 bit wide fields specifying the major and minor version numbers of the server, followed by a 16 bit field specifying the size of the remaining data in the packet.

The success code will either be 1 for success, or 0 for failure. The major and minor fields will be 11 and 0 respectively. The length field will be the size of the remaining data to be read, however, the value given is divided by 4, so to get the number of bytes left in the packet, you need to multiply the length field by 4.

This data can be read easily into a structure like the following:

```
struct x11_conn_reply
{
        uint8_t success;
        uint8_t pad1;
        uint16_t major,
                        minor;
        uint16_t length;
};
```

Then things start to get tricky. There is probably going to be almost 2000 bytes left to read. There are a few more sections left in the packet, but now that we know the size of the packet, we can read the rest of the packet into a buffer.

The next section we read will be the connection setup section. This data is a fixed size of 24 bytes, and it fits nicely into a structure like the following:

```
struct x11_conn_setup
{
        uint32_t release;
        uint32_t id_base,
                        id_mask;
        uint32_t motion_buffer_size;
        uint16_t vendor_length;
        uint16_t request_max;
        uint8_t roots;
        uint8_t formats;
        uint8_t image_order;
        uint8_t bitmap_order;
        uint8_t scanline_unit,
                        scanline_pad;
        uint8_t keycode_min,
                        keycode_max;
        uint32_t pad;
};
```

There are some very important fields in this section. The release is the actual version of the X server running, we can ignore this. The id_base and id_mask are very important. These determine what numbers we will be using for resource IDs in our application. The motion_buffer_size is of little importance for now. The vendor_length is important to remember, even though we probably do not care who the X server vendor is, nor should any application take any action based on the vendor, but we will need the length so we know how many bytes we need to burn after the section and before we get to important stuff. request_max is the maximum size a request can be. roots is the number of roots. formats is the number of available formats. image_order and bitmap_order are for endianness. scanline_unit and scanline_pad will be one of 8, 16, or 32. keycode_min and keycode_max are the minimum and maximum values possible for key codes. The padding at the end is just unused padding.

Immediately following this section is the vendor name. This is a variable length string, but we already know it's length because the connection setup section includes the *vendor_length* field, which will contain the length of the vendor name string in bytes. I have a feeling that this is never actually useful, so chances are you can ignore it, you might not even bother allocating memory for it.

After that, we will get a list of all the available pixmap formats. These are fixed length structures, but there is a variable number of them in the packet. However, we know how many there will be due to the *formats* field in the setup section. The structure for storing these pixmap formats looks like this:

```
struct x11_pixmap_format
{
        uint8_t depth;
        uint8_t bpp;
        uint8_t scanline_pad;
        uint8_t pad1;
        uint32_t pad2;
};
```

Inside this structure we get the color depth, bits per pixel, scanline padding, and 40 bits of unused padding. Remember that there will be more than just one available pixmap format. You will need to remember these if you want to draw pixmaps on the screen.

After the pixmap formats, we move on to the next section: root windows. If you have ever worked with X in the past, then you probably know what this means. The root window is the bottom most window, sometimes the one that displays the desktop. All other windows are a child of the root window. In order for you to draw your own window on the screen, you will need to know the ID of the root window. This is where we get it from.

I guess it is possible for there to be multiple root windows, based on how the protocol is designed, but I don't see why it would ever happen, but there could be something I am overlooking. Anyways, the number of root windows was defined in the setup section by the *roots* field. The structure of a root window looks like this:

```
struct x11_root_window
{
        uint32_t id;
        uint32_t colormap;
        uint32_t white,
                        black;
        uint32_t input_mask;
        uint16_t width,
                        height;
        uint16_t width_mm,
                        height_mm;
        uint16_t maps_min,
                        maps_max;
        uint32_t root_visual_iD;
        uint8_t backing_store;
        uint8_t save_unders;
        uint8_t depth;
        uint8_t depths;
};
```

Now, this is another place where things can get complicated programatically. Clearly, this is an important part of the handshake to interpret, however, there is a lot of stuff here. Also, let's not forget that it is technically possible (according to the protocol) for there to be more than one root window, which makes things even more complicated when we get into the depths. There will be multiple depths, and each depth will have a list of visuals to go with it. The structure for storing depths looks something like this:

```
struct x11_depth
{
        uint8_t depth;
        uint8_t pad1;
        uint16_t visuals;
        uint32_t pad2;
};
```

As you can see, the depth structure is very simple. However, what makes it complicated is that there will be a list of all valid visuals for that depth following each entry. The number of visuals for each depth is specified in the visuals field of the depth structure. A visuals structure looks like this:

```
struct x11_visual
{
        uint8_t group;
        uint8_t bits;
        uint16_t colormap_entries;
        uint32_t mask_red,
                         mask_green,
                         mask_blue;
        uint32_t pad;
};
```

It is probably helpful to keep all the information gathered from the handshake structure of its own. I like to call it x11_connection, because it makes sense to associate this information with the X connection.

```
int x11_handshake(int sock, struct x11_connection *conn)
{
        struct x11_conn_req req = {0};
        req.order='l';  //Little endian
        req.major=11; req.minor=0; //Version 11.0
        write(sock,&req,sizeof(struct x11_conn_req)); //Send request

        read(sock,&conn->header,sizeof(struct x11_conn_reply)); //Read reply header

        if (conn->header.success==0)
                return conn->header.success;

        conn->setup = sbrk(conn->header.length*4);      //Allocate memory for remainder of
data
        read(sock,conn->setup,conn->header.length*4);   //Read remainder of data
        void* p = ((void*)conn->setup)+sizeof(struct x11_conn_setup)+conn->setup-
>vendor_length; //Ignore the vendor
        conn->format = p;       //Align struct with format sections
        p += sizeof(struct x11_pixmap_format)*conn->setup->formats;     //move pointer to end
of section
        conn->root = p; //Align struct with root section(s)
        p += sizeof(struct x11_root_window)//move pointer to end of section
        conn->depth = p; //Align depth struct with first depth section
        p += sizeof(struct x11_depth);  //move pointer to end of section
        conn->visual = p; //Align visual with first visual for first depth

        return conn->header.success;
}
```

Note that this function places the pointers for root, depth, and visual only on the first instace of each section. That means that root[1] will not give you correct information about the second root window (if there ever is one), and depth[1] will only give you the second depth structure for the first root window, assuming it has more than one. Be aware of this.

As with XCB, the next thing that needs to be done after the connection is established, is to generate a resource ID so we can set up the graphics context. This doesn't actually require any communication with the server, all the information we need was part of the handshake. The connection setup section included the fields *id_base* and *id_mask*. To generate a resource ID, all we need to do is find an unused value above id_base that fits in the mask of id_mask. The easiest thing to do is have a function that keeps an the last ID used in an internal state variable, and just increment the value each time.

```
uint32_t x11_generate_id(struct x11_connection *conn)
{
        static uint32_t id = 0;
        return (conn->setup->id_mask & id++ | conn->setup->id_base);
}
```

Once we have an ID, things are going to start getting a lot more familiar for XCB programmers. One thing that we will need to deal with is op codes. In XCB, to create a graphics context, all you need to do is call the function: xcb_create_gc. However, we can't do that, since we aren't using XCB and we don't have any predefined functions. So to get the server to do what we want, we need to send it the correct op code for the operation we want to perform. The best way to handle this is to set them up as preprocessor constants, at least for the core operations (extensions are more complicated). Op codes 0-127 are used for the core protocol and are constant. Op codes about 127 are used for extensions, and are dynamically assigned. For now, we are only focusing on the core protocol.

I don't really want to list out all the op code in the core protocol, so I will just give you the ones that will be used in this tutorial as an example. A full listing of op codes can be found in the xproto tarball which you can get from x.org.

```
#define X11_OP_REQ_CREATE_WINDOW        0x01
#define X11_OP_REQ_MAP_WINDOW           0x08
#define X11_OP_REQ_CREATE_GC            0x37
```

Then, all we have to do is send a packet with the op code and the correct arguments for that op code. It is also possible to send multiple requests in one single packet, simply by concatenating the requests, which is exactly what XCB does with all requests until xcb_flush is called.

There are a few ways one to implement this in your application. One might be similar to how the XCB API works, with a function for each request. Or you might be able to get away with a variadic function to which you pass your op code as the first argument. However, not all arguments are of the same size in the X protocol, so you would need so way to determine this, either based on the op code passed to it, or to pass a size along with each argument. Unfortunately, both of those methods of implementing request via variadic functions has significant limitations, so I will show each request type as it's own function. For implementing functions other than the ones I will be showing in this tutorial, a good place to look is in the XCB API, XCB sources, XCB documentation, and packet captures.

One thing to note is that in X11, many requests have a flag field. This flag field tells the server how many additional fields will follow the flags field and what will be stored inside them. This means that many X11 requests are actually variable length. This can be annoying when trying to program efficiently. Fortunately, each field is either 32 bits wide, or padded out to 32 bits, and there are many algorithms available for counting bits, so we can determine the size of the request fairly easily. I will include an algorithm for counting bits in my example, the algorithm is adapted from a algorithm found on this page. Ideally, if you are only targeting new enough processors, then you could write a small ASM function to utilize the POPCNT instruction. However, my processor (Core 2 Duo) does not support this (SSE4.2). You might also use __builtin_popcount, assuming your compiler is GCC or clang and a standard c library is being used in your application.

```
//MIT HACKMEM bit counting algorithm
int count_bits(uint32_t n)
{
        unsigned int c;

        c = n - ((n >> 1) & 0x55555555);
        c = ((c >> 2) & 0x33333333) + (c & 0x33333333);
        c = ((c >> 4) + c) & 0x0F0F0F0F;
        c = ((c >> 8) + c) & 0x00FF00FF;
        c = ((c >> 16) + c) & 0x0000FFFF;

        return c;
}
```

Once you have a way to count set bits easily, then we are one step closer to being able to draw a window. However, we will also need to be able to set the flags that we want to have set. We could memorize all the numbers for each flag, or we could use the preprocessor. Love the preprocessor.

```
#define X11_FLAG_GC_FUNC 0x00000001
#define X11_FLAG_GC_PLANE 0x00000002
#define X11_FLAG_GC_BG 0x00000004
#define X11_FLAG_GC_FG 0x00000008
#define X11_FLAG_GC_LINE_WIDTH 0x00000010
#define X11_FLAG_GC_LINE_STYLE 0x00000020
#define X11_FLAG_GC_FONT 0x00004000
...
#define X11_FLAG_WIN_BG_IMG 0x00000001
```

```
#define X11_FLAG_WIN_BG_COLOR 0x00000002

#define X11_FLAG_WIN_BORDER_IMG 0x00000004

#define X11_FLAG_WIN_BORDER_COLOR 0x00000008

#define X11_FLAG_WIN_EVENT 0x00000800
```

Once we know the right flags, and we know the right op codes, we just need to know the right sizes and orders of the arguments before we can start sending requests. To find the correct sizes and orders of arguments, we can look either at packet captures or the X11 documentation: Requests section. Though packet captures are easier to read and understand, the documentation is more thorough, once you figure out its little nuances.

Once you have figured out the right order, you can write a nice function that will take care of sending out the right data. For creating a graphics context, your code might look like the following:

```
void x11_create_gc(int sock, struct x11_connection *conn, uint32_t id, uint32_t target,
uint32_t flags, uint32_t *list)
{
        uint16_t flag_count = count_bits(flags);

        uint16_t length = 4 + flag_count;
        uint32_t *packet = sbrk(length*4);

        packet[0]=X11_OP_REQ_CREATE_GC | length<<16;
        packet[1]=id;
        packet[2]=target;
        packet[3]=flags;
        int i;
        for (i=0;i<flag_count;i++)
                packet[4+i] = list[i];

        write(sock,packet,length*4);

        sbrk(-(length*4));

        return;
}
```

In this function, I pass in both the socket to write to, and the connection structure. This is not necessary really because the structure is not actually used in the function. It would also make sense to keep the socket file descriptor in a field of the connection structure, however, I did not do this in my code.

As you can probably see, all this does is put the arguments in the correct order and write them to the socket. The one thing to note in this function is the flags field. There are other ways that this could be set up to work, but what I have done is passed in a pointer to the start of the option data. The count_bits function determines how many 32 bit numbers should be read from the list and adds them to the end of the packet. This is very similar to what XCB does in the same cases.

The function for creating a window looks very similar:

```
void x11_create_win(int sock, struct x11_connection *conn, uint32_t id, uint32_t parent,
                                        uint16_t x, uint16_t y, uint16_t w, uint16_t h,
                                        uint16_t border, uint16_t group, uint32_t visual,
                                        uint32_t flags, uint32_t *list)
{
        uint16_t flag_count = count_bits(flags);

        uint16_t length = 8 + flag_count;
        uint32_t *packet = sbrk(length*4);

        packet[0]=X11_OP_REQ_CREATE_WINDOW | length<<16;
        packet[1]=id;
        packet[2]=parent;
        packet[3]=x | y<<16;
        packet[4]=w | h<<16;
        packet[5]=border<<16 | group;
        packet[6]=visual;
        packet[7]=flags;
        int i;
        for (i=0;i<flag_count;i++)
                packet[8+i] = list[i];

        write(sock,packet,length*4);

        sbrk(-(length*4));
        return;
}
```

As you can see, creating a window is just about as simple as creating a graphics context. It just has a few more parameters. You also have to take note of the fact that the x,y and w,h arguments are 16 bits wide and not 32 bits.

Mapping a window to the screen is quite a bit simpler than the previous requests:

```
void x11_map_window(int sock, struct x11_connection *conn, uint32_t id)
{
        uint32_t packet[2];
        packet[0]=X11_OP_REQ_MAP_WINDOW | 2<<16;
        packet[1]=id;
        write(sock,packet,8);

        return;
}
```

Mapping a window doesn't require a flags argument or a list of optional arguments. There is only one argument, the window ID.
Once the window is mapped to the screen, you should be able to see it show up on the screen. If you create a window with the background flag set and a background color set up, you can see that color in the background of the window.

Now, once you have a nice window finally displaying on your screen after all that work, let's talk about drawing something in the window.

Unlike the other methods and protocols described in this tutorial, using X11 works by communicating with a server (as I REALLY hope you realized already), as opposed to just writing your graphics into a framebuffer. This means that any time you want to draw something on the screen, you will have to actually send that something to the server. X11 gives us two ways of doing this, we can send an image, or we can instruct the server to draw primatives.

Although this tutorial is more concerned with drawing primatives, I want to actually talk about drawing images instead. The reason for this is that X11 already includes requests for drawing primatives, which are easy to figure out and use, and because all of the other graphical display methods in this tutorial center around drawing onto a framebuffer, and a framebuffer is essentially just an image already. So therefore, in order to apply the techniques learned from other graphical display methods in this tutorial, you will probably want to be drawing images. Also, using only primatives will impose limitations later when you do decide that you want to start drawing actual pictures loaded from files.

Displaying images is a little bit more difficult than the other requests we covered already. To display a window, we told the server to create a window and draw it on the screen, and optionally to fill the window with a color. However, to display an image, we have to have the image client side and then transfer that entire image up to the server. This is accomplished with the "put image" request. The put image request (or x11_put_img as I call it) will send the image data up to the server and tell the server where to put it. The request requires a "drawable" argument, which is the location where the server will put the image. This could be pretty much any drawable on the server. We could, for example, simply tell the server to jam that image directly onto the window we created, and it will work fine. However, that would mean that every time we wanted to do something with that image, we would have to re-upload it to the server, which is an unacceptable overhead cost in most cases. To compensate for this, we can create a "pixmap" on the server, then put our image into that pixmap (which is a drawable), and then map that pixmap onto our window where and when we want it. That method has the advantage of keeping that image in server memory so that it can be retreived at any time without uploading the data again. Using a pixmap will probably be the preferred method most of the time, however, sending the image data directly to your window is simpler, and you should be able to figure out how to create and draw to a pixmap on your own since it is very similar to other things that we have already done in this tutorial, so I will only show how to draw directly on the created window.

Basically, sending an image is the same as any other request, it just includes one really big field at the end containing the image data. To send an image to the server, you should use the "put image" request. This request needs a drawable (where to draw it) an a GC. In addition, it also requires a height, width, and x and y. The height and width are the size of the image, these are important because these help the server calculate how big the image will be in conjunction with the depth. In order for the server to accept and draw the image, the depth must be the same as the GC used to draw, otherwise the X server will send an error message and draw nothing. The size must also be correct. To calculate the size, simply multiply the height and width of the image, then multiply by the depth (32 = 32 bits) rounded to the nearest boundary (depth 24 = 32 / depth 16 = 16 / depth 8 = 8 / depth < 8 = 8). Following the normal fields in the request is one large data field, which contains the image data. Implementing this function might look something like this:

```
#define X11_IMG_FORMAT_MONO 0x0
#define X11_IMG_FORMAT_XY 0x01
#define X11_IMG_FORMAT_Z 0x02
void x11_put_img(int sock, struct x11_connection *conn, uint8_t format, uint32_t target,
uint32_t gc,
                                uint16_t w, uint16_t h, uint16_t x, uint16_t y, uint8_t depth,
void *data)
```

```
{
        uint32_t packet[6];
        uint16_t length = ((w * h)) + 6;

        packet[0]=X11_OP_REQ_PUT_IMG | format<<8 | length<<16;
        packet[1]=target;
        packet[2]=gc;
        packet[3]=w | h<<16;
        packet[4]=x | y<<16;
        packet[5]=depth<<8;

        write(sock,packet,24);
        write(sock,data,(w*h)*4);

        return;
}
```

Notice that I write the data with a seperate system call. Since the connection uses a stream socket, this still works with no issues, there is no distinction between packets in TCP or Unix Domain stream sockets. This also allows us to avoid copying image data into a new buffer before writing it to the socket. Note that we calculated the size of the image by multiplying height and width, this will not work all the time because it assumes a depth of either 24 or 32 bits (which is what I expect on modern systems).

And now, without further ado, a full, working example:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <fcntl.h>                                      //O_RDONLY
#include <sys/types.h>                          //uint64_t
#include <sys/socket.h>                         //Socket related constants
#include <sys/un.h>                                     //Unix domain constants
#include <netinet/in.h>                         //Socket related constants

#ifdef __linux__
        #include <linux/limits.h>               //PATH_MAX
```

```
#else
        #define PATH_MAX 4096
#endif

#define X11_OP_REQ_CREATE_WINDOW        0x01
#define X11_OP_REQ_MAP_WINDOW           0x08
#define X11_OP_REQ_CREATE_PIX           0x35
#define X11_OP_REQ_CREATE_GC            0x37
#define X11_OP_REQ_PUT_IMG                      0x48

struct x11_conn_req
{
        uint8_t order;
        uint8_t pad1;
        uint16_t major,
                        minor;
        uint16_t auth_proto,
                        auth_data;
        uint16_t pad2;
};

struct x11_conn_reply
{
        uint8_t success;
        uint8_t pad1;
        uint16_t major,
                        minor;
        uint16_t length;
};

struct x11_conn_setup
{
        uint32_t release;
        uint32_t id_base,
                        id_mask;
        uint32_t motion_buffer_size;
        uint16_t vendor_length;
        uint16_t request_max;
        uint8_t roots;
        uint8_t formats;
        uint8_t image_order;
        uint8_t bitmap_order;
        uint8_t scanline_unit,
                scanline_pad;
        uint8_t keycode_min,
                keycode_max;
        uint32_t pad;
};

struct x11_pixmap_format
{
        uint8_t depth;
        uint8_t bpp;
        uint8_t scanline_pad;
        uint8_t pad1;
        uint32_t pad2;
};

struct x11_root_window
{
        uint32_t id;
        uint32_t colormap;
        uint32_t white,
                black;
        uint32_t input_mask;
        uint16_t width,
                height;
        uint16_t width_mm,
                height_mm;
        uint16_t maps_min,
                maps_max;
        uint32_t root_visual_id;
        uint8_t backing_store;
        uint8_t save_unders;
        uint8_t depth;
        uint8_t depths;
};

struct x11_depth
{
        uint8_t depth;
        uint8_t pad1;
        uint16_t visuals;
        uint32_t pad2;
};

struct x11_visual
{
        uint8_t group;
        uint8_t bits;
        uint16_t colormap_entries;
        uint32_t mask_red,
                        mask_green,
                        mask_blue;
        uint32_t pad;
};

struct x11_connection
{
        struct x11_conn_reply header;
        struct x11_conn_setup *setup;
        struct x11_pixmap_format *format;
        struct x11_root_window *root;
        struct x11_depth *depth;
        struct x11_visual *visual;
};

struct x11_error
```

```
{
        uint8_t success;
        uint8_t code;
        uint16_t seq;
        uint32_t id;
        uint16_t op_major;
        uint8_t op_minor;
        uint8_t pad[21];
};

//Copy characters from one string to another until end
int strcopy(char *dest, const char *src, char end)
{
        int i;
        for (i=0; src[i]!=end; i++)
                dest[i]=src[i];
        return i;
}

//Copy n characters from one string to another
char *strncopy(char *dest, const char *src, size_t n)
{
        int i;
        for (i=0; i<n; i++)
                dest[i]=src[i];
        return dest;
}

int count_bits(uint32_t n)
{
        unsigned int c;

        c = n - ((n >> 1) & 0x55555555);
        c = ((c >> 2) & 0x33333333) + (c & 0x33333333);
        c = ((c >> 4) + c) & 0x0F0F0F0F;
        c = ((c >> 8) + c) & 0x00FF00FF;
        c = ((c >> 16) + c) & 0x0000FFFF;

        return c;
}

void error(char *msg, int len)
{
        write(0, msg, len);
        exit(1);

        return;
}

int x11_handshake(int sock, struct x11_connection *conn)
{
        struct x11_conn_req req = {0};
        req.order='l';  //Little endian
        req.major=11; req.minor=0; //Version 11.0
        write(sock,&req,sizeof(struct x11_conn_req)); //Send request

        read(sock,&conn->header,sizeof(struct x11_conn_reply)); //Read reply header

        if (conn->header.success==0)
                return conn->header.success;

        conn->setup = sbrk(conn->header.length*4);      //Allocate memory for remainder of
data
        read(sock,conn->setup,conn->header.length*4);   //Read remainder of data
        void* p = ((void*)conn->setup)+sizeof(struct x11_conn_setup)+conn->setup-
>vendor_length; //Ignore the vendor
        conn->format = p;       //Align struct with format sections
        p += sizeof(struct x11_pixmap_format)*conn->setup->formats;     //move pointer to end
of section
        conn->root = p; //Align struct with root section(s)
        p += sizeof(struct x11_root_window)*conn->setup->roots; //move pointer to end of
section
```

```
        conn->depth = p; //Align depth struct with first depth section
        p += sizeof(struct x11_depth);  //move pointer to end of section
        conn->visual = p; //Align visual with first visual for first depth

        return conn->header.success;
}

uint32_t x11_generate_id(struct x11_connection *conn)
{
        static uint32_t id = 0;
        return (id++ | conn->setup->id_base);
}


#define X11_FLAG_GC_FUNC 0x00000001
#define X11_FLAG_GC_PLANE 0x00000002
#define X11_FLAG_GC_BG 0x00000004
#define X11_FLAG_GC_FG 0x00000008
#define X11_FLAG_GC_LINE_WIDTH 0x00000010
#define X11_FLAG_GC_LINE_STYLE 0x00000020
#define X11_FLAG_GC_FONT 0x00004000
#define X11_FLAG_GC_EXPOSE 0x00010000
void x11_create_gc(int sock, struct x11_connection *conn, uint32_t id, uint32_t target,
uint32_t flags, uint32_t *list)
{
        uint16_t flag_count = count_bits(flags);

        uint16_t length = 4 + flag_count;
        uint32_t *packet = sbrk(length*4);

        packet[0]=X11_OP_REQ_CREATE_GC | length<<16;
        packet[1]=id;
        packet[2]=target;
        packet[3]=flags;
        int i;
        for (i=0;i<flag_count;i++)
```

```
                packet[4+i] = list[i];

        write(sock,packet,length*4);

        sbrk(-(length*4));

        return;
}

#define X11_FLAG_WIN_BG_IMG 0x00000001
#define X11_FLAG_WIN_BG_COLOR 0x00000002
#define X11_FLAG_WIN_BORDER_IMG 0x00000004
#define X11_FLAG_WIN_BORDER_COLOR 0x00000008
#define X11_FLAG_WIN_EVENT 0x00000800
void x11_create_win(int sock, struct x11_connection *conn, uint32_t id, uint32_t parent,
                                        uint16_t x, uint16_t y, uint16_t w, uint16_t h,
                                        uint16_t border, uint16_t group, uint32_t visual,
                                        uint32_t flags, uint32_t *list)
{
        uint16_t flag_count = count_bits(flags);

        uint16_t length = 8 + flag_count;
        uint32_t *packet = sbrk(length*4);

        packet[0]=X11_OP_REQ_CREATE_WINDOW | length<<16;
        packet[1]=id;
        packet[2]=parent;
        packet[3]=x | y<<16;
        packet[4]=w | h<<16;
        packet[5]=border<<16 | group;
        packet[6]=visual;
        packet[7]=flags;
        int i;
        for (i=0;i<flag_count;i++)
                packet[8+i] = list[i];
        write(sock,packet,length*4);

        sbrk(-(length*4));

        return;
}

void x11_map_window(int sock, struct x11_connection *conn, uint32_t id)
{
        uint32_t packet[2];
        packet[0]=X11_OP_REQ_MAP_WINDOW | 2<<16;
        packet[1]=id;
        write(sock,packet,8);

        return;
}

#define X11_IMG_FORMAT_MONO 0x0
#define X11_IMG_FORMAT_XY 0x01
#define X11_IMG_FORMAT_Z 0x02
void x11_put_img(int sock, struct x11_connection *conn, uint8_t format, uint32_t target,
uint32_t gc,
                                uint16_t w, uint16_t h, uint16_t x, uint16_t y, uint8_t depth,
void *data)
{
        uint32_t packet[6];
        uint16_t length = ((w * h)) + 6;

        packet[0]=X11_OP_REQ_PUT_IMG | format<<8 | length<<16;
        packet[1]=target;
        packet[2]=gc;
        packet[3]=w | h<<16;
        packet[4]=x | y<<16;
        packet[5]=depth<<8;

        write(sock,packet,24);
        write(sock,data,(w*h)*4);


        return;
}


int main(int argc, char **argv)
{
        int sockfd, clilen, srv_len, openfd;


        if (argc==1)
        {
                struct sockaddr_un serv_addr={0};


                //Create the socket
                sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
                if (sockfd < 0)
                        error("Error opening socket\n",21);


                serv_addr.sun_family = AF_UNIX;
                strcopy(serv_addr.sun_path, "/tmp/.X11-unix/X0", 0);
                srv_len = sizeof(struct sockaddr_un);


                connect(sockfd,(struct sockaddr *)&serv_addr,sizeof(serv_addr));
        }
        else
        {
                struct sockaddr_in serv_addr={0};


                //Create the socket
                sockfd = socket(AF_INET, SOCK_STREAM, 0);
                if (sockfd < 0)
                        error("Error opening socket\n",21);


                serv_addr.sin_family = AF_INET;
                serv_addr.sin_addr.s_addr=0x0100007f;
                serv_addr.sin_port = htons(6001);

```

```
		connect(sockfd,(struct sockaddr *)&serv_addr,sizeof(serv_addr));
	}

	struct x11_connection conn = {0};
	x11_handshake(sockfd,&conn);

	int gc = x11_generate_id(&conn);
	uint32_t val[32];
	val[0]=0x00aa00ff;
	val[1]=0x00000000;
	x11_create_gc(sockfd,&conn,gc,conn.root[0].id,X11_FLAG_GC_BG|X11_FLAG_GC_EXPOSE,val);

	int win = x11_generate_id(&conn);
	val[0]=0x00aa00ff;

x11_create_win(sockfd,&conn,win,conn.root[0].id,200,200,400,200,1,1,conn.root[0].root_visual_id,X11


	x11_map_window(sockfd,&conn,win);

	uint32_t data[1600]={0};
	int i;
	for (i=0;i<1600;i++)
		data[i]=0x0055ff33;

	x11_put_img(sockfd,&conn,X11_IMG_FORMAT_Z,win,gc,40,40,0,0,conn.root[0].depth,data);

	while (1) ;
}
```

Wayland (Coming Soon)

I have been working on this, but it will take time, be patient.

Mir (Coming Soon)

Once I get Mir actually compiled and running on my computer, then I will start work on actually developing native Mir applications. This will probably take the longest.

Useful Drawing Concepts

Now that you have learned how to actually draw on the screen. You will probably want to learn how to do more interesting and useful things with your new knowledge. Because we are working with such low level graphics systems, we do not automatically get things like double buffering and primatives, you have to implement that stuff on your own. This section may help.

Double Buffering

Double buffering is very simple, and can have a profound impact on your graphics. With only a single video buffer, the contents of the buffer might be displayed at a bad time, such as right after you have cleared the screen, but before anything has been redrawn. This makes your graphics look pretty ugly. It will look like your screen is flickering and probably isn't great for your eyes.

Double buffering solves this problem by never presenting anything to the screen until it is fully drawn. The basic idea is that instead of having only one buffer, there are two buffers of equal size: a front buffer, and a back buffer. The front buffer is the one being drawn on the screen. Nothing should be drawn onto the front buffer. The back buffer is not displayed, it is just memory. Everything that should be displayed is first drawn into the back buffer. Then, once the entire frame is ready, the buffers are swapped, and the back buffer becomes the new front buffer. This way, the front buffer always contains a complete frame.

Implementing double buffering is easy. The first thing you need to do is allocate a second buffer, the exact same size as your front buffer. For instance, if we are working with the Linux framebuffer device, we might use the following code.

```
static uint32_t *back_buffer;


void init_back_buffer()
{
        back_buffer = (uint8_t*)mmap(0, vinfo.yres_virtual * finfo.line_length,
PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, (off_t)0);
}
```

Then, instead of drawing into the front buffer, all your drawing functions must draw into the back buffer instead. So, if we wanted to make the whole screen purple, we might use the following code (compare to example in fbdev section).

```
        for (x=0;x<vinfo.xres;x++)
                for (y=0;y<vinfo.yres;y++)
                {
                        long location = (x+vinfo.xoffset) * (vinfo.bits_per_pixel/8) +
(y+vinfo.yoffset) * finfo.line_length;
                        *((uint32_t*)(back_buffer + location)) = pixel_color(0xFF,0x00,0xFF,
&vinfo);
                }
```

We can draw as much as we want into the back buffer, but it will never be displayed on the screen, until we swap it with the front buffer that is.

The way we swap buffers is going to be dependent on the method we are using to render to the screen. For instance, DRM/KMS provides it's own method of swapping buffers. You might be able to just tell your video driver to use your back buffer as the front buffer. If all else fails, you can just copy the contents of the entire back buffer onto the front buffer. In this case, we do not need to actually "swap" the buffers since we do not really care about moving to contents of the front buffer into the back buffer, so we only need to copy the back buffer to the front buffer and be done with it. We can use this method of swapping buffers using the fbdev rendering method. The code to swap buffers might look like this.

```
inline void swap_buffers()
{
        int i;
        for (i=0;i<(vinfo.yres_virtual * finfo.line_length)/4;i++)
        {
                ((uint32_t*)(fbp))[i] = back_buffer[i];
        }
}
```

Please note that there is a lot of room for optimization in this code. On systems with SIMD instructions, such x86_64 (SSE, SSE2, SSE3, SSE4.1, etc..) multiple bytes of the buffer can be copied at once. Writing this routine in assembly language might be helpful.

However, there is even a better way to do double buffer using fbdev. Although the interface does not provide a way to switch buffers, we can use a little trick to simulate this behavior.

The fbdev driver allows for what it calls "panning", where the framebuffer can be bigger than the screen and then the driver can be instructed to draw the correct portion of the buffer on the screen. To simulate buffer switching behavior, we can allocate both buffers as one big buffer and then draw in the one that is off the screen (our back buffer), and then tell the device to "pan" to the offscreen portion of the buffer. We tell the device to pan with the FBIOPAN_DISPLAY ioctl.

```
uint8_t *fbp,   //Front buffer base pointer
                *bbp;   //back buffer base pointer


void init_fbdev()
{
        ...


        fbp = mmap(0, screensize*2, PROT_READ | PROT_WRITE, MAP_SHARED, fb_fd, (off_t)0);
        bbp = fbp + screensize;
}


void clear()
{
        for (x=0;x<vinfo.xres;x++)
                for (y=0;y<vinfo.yres;y++)
                {
                        long location = (x+vinfo.xoffset) * (vinfo.bits_per_pixel/8) +
(y+vinfo.yoffset) * finfo.line_length;
                        *((uint32_t*)(bbp + location)) = pixel_color(0xFF,0x00,0xFF, &vinfo);
                }
}


inline void swap_buffers()
{
        if (vinfo.yoffset==0)
                vinfo.yoffset = screensize;
        else
                vinfo.yoffset=0;


        //"Pan" to the back buffer
        ioctl(fb_fd, FBIOPAN_DISPLAY, &vinfo);


        //Update the pointer to the back buffer so we don't draw on the front buffer
        long tmp;
        tmp=fbp;
        fbp=bbp;
        bbp=tmp;
}
```

That's all you need to know to do double buffering. Make sure you remember to swap your buffers when you want your next frame to be displayed.

Drawing Primatives

You will probably need to draw shapes on the screen if you are going to do graphics. Drawing single pixels isn't all that useful for most applications. Here I describe a few common shape drawing algorithms that execute very quickly and do what you need. I have used most of these in my own code.
However, I can't cover everything, but there are lots of other resources online that can help. Here is one very good one: Primitive Shapes & Lines.

Lines

Drawing lines is very important. Probably every single graphical application draws lines for something or another. Many programs might only need to draw horizontal and vertical lines, while other applications, such as games, will need to draw lines of an arbitrary angle.

Drawing horizontal lines and vertical lines is by far the simplest. You can draw these types of lines just by iterating over the x or y values and plotting each pixel. The following code shows these two functions. Please note that these functions make the assumption that x1 is less than x2 and y1 is less than y2.

```
void draw_horizontal_line(int x1, int x2, int y, uint32_t pixel)
{
        int i;
        for (i=x1;i<x2;i++)
                draw(i,y,pixel);
}
void draw_vertical_line(int x, int y1, int y2, uint32_t pixel)
{
        int i;
        for (i=y1;i<y2;i++)
                draw(x,i,pixel);
}
```

These functions are very simple, about as simple as possible. However, it should be noted that there is still room for optimization, especially in the horizontal line version. If we are using a framebuffer stored as a contiguous block of main memory, then we know that our horizontal line is also stored as a contiguous block of memory, and we can determine it's size by subtracting x1 from x2. We can then copy memory into that block instead of changing each pixel one at a time. Machines with SIMD instructions (e.g. SSE, SSE2, etc...) will be particularly adept at this type of operation, it might be useful to write this portion of your code in assembly. However, your compiler's optimizer should also be able to do a pretty good job of optimizing this code on it's own.

For more complicated lines, we can use an algorithm known as Bresenham's line algorithm. It's basically the fastest one available, and it works great. You aren't going to get any super fancy features like anti-aliasing, but in most cases, this is exactly what you want. This code has been adapted from code found on this page: Primative Shapes & Lines written in 1996.

```
void draw_line(int x1, int y1, int x2, int y2, uint32_t pixel)
{
        int i,dx,dy,sdx,sdy,dxabs,dyabs,x,y,px,py;

        dx=x2-x1;                       //Delta x
        dy=y2-y1;                       //Delta y
        dxabs=abs(dx);          //Absolute delta
        dyabs=abs(dy);          //Absolute delta
        sdx=(dx>0)?1:-1; //signum function
        sdy=(dy>0)?1:-1; //signum function
        x=dyabs>>1;
        y=dxabs>>1;
        px=x1;
        py=y1;

        if (dxabs>=dyabs)
        {
                for(i=0;i<dxabs;i++)
                {
                        y+=dyabs;
                        if (y>=dxabs)
                        {
                                y-=dxabs;
                                py+=sdy;
                        }
                        px+=sdx;
                        draw(px,py,pixel);
                }
        }
        else
        {
                for(i=0;i<dyabs;i++)
                {
                        x+=dxabs;
                        if (x>=dyabs)
                        {
                                x-=dyabs;
                                px+=sdx;
                        }
                        py+=sdy;
                        draw(px,py,pixel);
                }
        }
}
```

Circles

Drawing a circle is pretty easy. Circles are fun because they are very uniform. If you divide a circle in half down the middle, each side is just a mirror of the other side. so for any point on the circle (x,y), the other side is just (-x,y). The same can be said if you divide in half across the center. Top and bottom are also just mirrors of each other. So for any one point on a circle, we actually know 4 points: (x,y) (-x,y) (x,-y) and (-x,-y). Then if we divide the circle again diagonally, we can calculate 4 more points by just exchanging x and y. So for every point we calculate, we can draw 8 points on the circle, which means we only need to find 1/8th of the circle in order to draw the whole thing.

The following code is an implementation of what is called the "midpoint circle algorithm".

```
//Draw a circle at (cx,cy)

void draw_circle(double cx, double cy, int radius, uint32_t pixel)

{

        inline void plot4points(double cx, double cy, double x, double y, uint32_t pixel)

        {

                draw(cx + x, cy + y,pixel);

                draw(cx - x, cy + y,pixel);

                draw(cx + x, cy - y,pixel);

                draw(cx - x, cy - y,pixel);

        }


        inline void plot8points(double cx, double cy, double x, double y, uint32_t pixel)

        {

                plot4points(cx, cy, x, y,pixel);

                plot4points(cx, cy, y, x,pixel);

        }


        int error = -radius;

        double x = radius;

        double y = 0;


        while (x >= y)

        {

                plot8points(cx, cy, x, y, pixel);


                error += y;

                y++;

                error += y;


                if (error >= 0)

                {

                        error += -x;

                        x--;

                        error += -x;
                }
        }
}
```

This algorithm is wonderful and simple and also very fast. I recommend using this algorithm any time you want to draw a circle. However, this algorithm only draws the outline of the circle, it does not fill the circle in. Fortunately, if you want to draw a filled circle instead, the modification is very easy. All you need to do is instead of plotting points (x,y) and (-x,y), draw a line between those points. This line drawing will always be horizontal, so the line drawing algorithm can be optimized accordingly (described in previous section). Here is the code (this code replaces plot4points function in the circle drawing algorithm above).

```
        inline void plot4points(double cx, double cy, double x, double y, uint32_t pixel)
        {
                draw_horizontal_line(cx + x, cx - x, cy + y,pixel);
                draw_horizontal_line(cx + x, cx - x, cy - y,pixel);
        }
```