School of Computing
FACULTY OF ENGINEERING AND
PHYSICAL SCIENCES

Final Report

Graphics in Xv6

Raka Alifiar Gunarto

Submitted in accordance with the requirements for the degree of BSc Computer Science

2021/22

COMP3931 Individual Report

The candidate confirms that the following have been submitted.

Items	Format	Recipient(s) and Date
Final Report	PDF file	PDF uploaded to Minerva (28/04/22)
Link to online code repository	URL	Sent to supervisor and assessor (28/04/22)
Link to demo video	URL	Sent to supervisor and assessor (28/04/22)

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of Student) Raka Gunarto

Summary

This project implements graphics functionality in Xv6. To do this, the following was developed: PCI enumeration, a basic VGA driver, a kernel grid window manager, and finally a user mode graphics library to interact with the windowing system. This report documents the development of these functionalities.

Acknowledgements

I would like to thank my supervisor Dr. Samuel Wilson for his guidance and support.

Contents

Chapter 1

Introduction and Background Research

1.1 Introduction

Operating systems (OSes) are specialised, low level software that manage a computer's resources and provides a common interface for programs. The most common operating systems used by most consumers today include MacOS and Windows. Xv6 is an operating system written by MIT for the purposes of teaching, which runs on the RISC-V architecture [1][2]. As it is meant to be a teaching OS, it has very barebones functionality, which includes a command line shell and some basic user programs by default.

This project aims to extend Xv6 with some graphics functionality.

1.2 Background research

1.2.1 Xv6

As mentioned in Chapter 1.1, Xv6 is a barebones OS used for teaching purposes developed by MIT. Its features include:

- Based on UNIX v6 (hence the v6 in Xv6)
- Written for 64 bit RISC-V architecture
- Multicore support with a round robin scheduler
- Runs on QEMU (qemu-system-riscv64)

1.2.2 QEMU

QEMU is an emulator that allows running a guest operating system that may run on a different target hardware platform. In our case we will be running Xv6, which runs on the RISC-V architecture, in QEMU for development and testing.

It also provides a way to debug the kernel through a GDB serial port, which greatly eases development and testing as we can connect a graphical debugger to the kernel.

We will also be using its device emulation to emulate a VGA card on the PCI bus for our driver to communicate with to display graphics. It offers many options each with its own pros and cons ranging from a simple VGA device with a linear framebuffer to cards with hardware acceleration. Additionally, QEMU also provides a simulated display that can be viewed through a VNC client, so we can see the graphics output of Xv6.

1.2.3 VGA

VGA (Video Graphics Array) has a well defined but complex hardware standard. Although complex, there is an abundance of technical resources [3] [4] [5] that can assist in developing a VGA driver. In addition, QEMU can emulate a VGA graphics card.

1.2.4 PCI

PCI (Peripheral Component Interconnect) is a standard that allows the communication of external hardware peripherals. This is how we will interface with the emulated VGA card to display graphics.

1.2.5 Hardware interfacing with Xv6

To find the emulated VGA card connected to system, we can walk the PCI configuration space and search for the Device ID and Vendor ID of the card. Usually the preferred method would be to walk the device tree [6] to find the hardware configuration and memory mapped addresses, but since we are running on the QEMU 'virt' machine we can hardcode these addresses for now.

```
static const MemMapEntry virt_memmap[] = {
    [VIRT_DEBUG] =        {        0x0,         0x100 },
    [VIRT_MROM] =         {     0x1000,         0xf000 },
    [VIRT_TEST] =         {   0x100000,         0x1000 },
    [VIRT_RTC] =          {   0x101000,         0x1000 },
    [VIRT_CLINT] =        {  0x2000000,        0x10000 },
    [VIRT_ACLINT_SSWI] =  {  0x2F00000,         0x4000 },
    [VIRT_PCIE_PIO] =     {  0x3000000,        0x10000 },
    [VIRT_PLIC] =         {  0xc000000, VIRT_PLIC_SIZE(VIRT_CPUS_MAX * 2) },
    [VIRT_UART0] =        { 0x10000000,         0x100 },
    [VIRT_VIRTIO] =       { 0x10001000,         0x1000 },
    [VIRT_FW_CFG] =       { 0x10100000,          0x18 },
    [VIRT_FLASH] =        { 0x20000000,     0x4000000 },
    [VIRT_PCIE_ECAM] =    { 0x30000000,    0x10000000 },
    [VIRT_PCIE_MMIO] =    { 0x40000000,    0x40000000 },
    [VIRT_DRAM] =         { 0x80000000,           0x0 },
};
```

Listing 1: /hw/riscv/virt.c:45-61 from the QEMU source code [7]

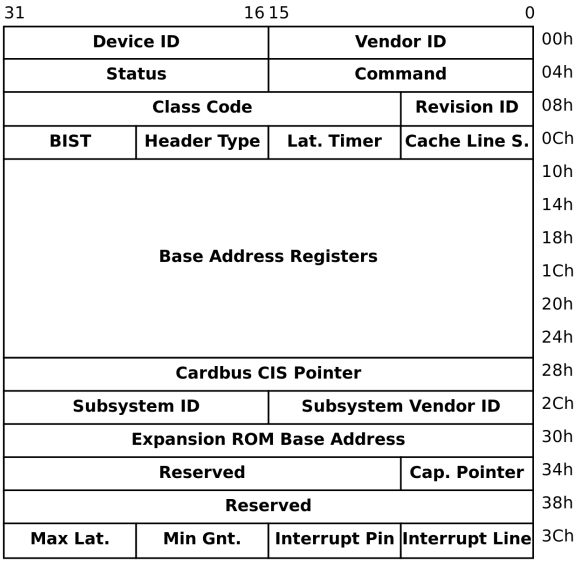

Figure 1.1: PCI Configuration Space Header Format [8]

As shown in listing 1, the QEMU 'virt' machine has defined mappings for the PCI configuration space, memory mapped IO and port IO (`[VIRT_PCIE_ECAM]`, `[VIRT_PCIE_MMIO]`, `[VIRT_PCIE_PIO]` respectively). Using this information, and the configuration space layout described by figure 1.1, finding the card by walking the configuration space table searching for the corresponding IDs should be trivial. Setting up and communicating with the card can be achieved by using the Base Address Registers in the configuration space to set the MMIO addresses for the framebuffer and the port IO to configure the card as documented in [4] can be done through the memory mapping for `[VIRT_PCIE_PIO]`.

1.2.6 Window managers

A window manager is required for multiple user space programs to share the screen. Otherwise concurrent graphical user programs would try to write to the same display and overwrite each other.

Windows

Windows provides a windowing system that provides free floating windows that can be created and managed using the Windows API [9]. It is unclear how much of the display server and compositor is running in user mode or kernel mode.

Linux - X Windowing System

The X windowing system also provides free floating windows that is managed by the X Display Server. The most commonly used implementation is the open source X.Org project [10]. It is a userspace and unprivileged display server that communicates with the kernel through display drivers.

Our implementation

We'll be taking inspiration from both implementations of windowing systems. For simplicity, since Xv6 does not have any mouse implementations, a grid window system is a good alternative that can use hotkeys for window switching. The window manager will sit in the kernel providing very basic functionality such as window allocation, input event queuing, and copying user data on the framebuffer (with bounds checking). This will reduce the attack surface on the kernel by having the rendering done in unprivileged user mode.

Additionally, we will be exposing the windows as device files, following the Unix mantra "everything is a file". This has the advantage of being able to use the included read and write system calls for communicating with the kernel window manager.

1.2.7 Development tools

Compile toolchain

Ubuntu has the tools required for Xv6 compilation in its 'apt' software repository. To ensure a consistent development environment, Docker will be used to create a development container with all the required tools (see Appendix C.1)

Debugging with GDB

As QEMU provides a serial GDB port to debug the kernel, we will be using GDB to attach to QEMU to inspect the kernel. GDB can be used to read the debugging symbols, which greatly eases debugging by allowing us to step through the kernel line by line and set breakpoints at desired lines. GDB can show the contents of variables during program execution with their original names. QEMU also supports interrupting the kernel via GDB to stop at its current instruction and enter step-through debugging, which will be an invaluable tool for infinite loops or deadlocks in the kernel.

IDE

Visual Studio Code [11] is a text editor that supports extensions which can have IDE features, making it suitable for many use cases. For our use case, it has support for Docker containers and a GUI for GDB (figure 1.2). With its 'tasks' functionality as well, VSCode can be configured to build xv6, run qemu and attach gdb in one shortcut, which will greatly streamline development.

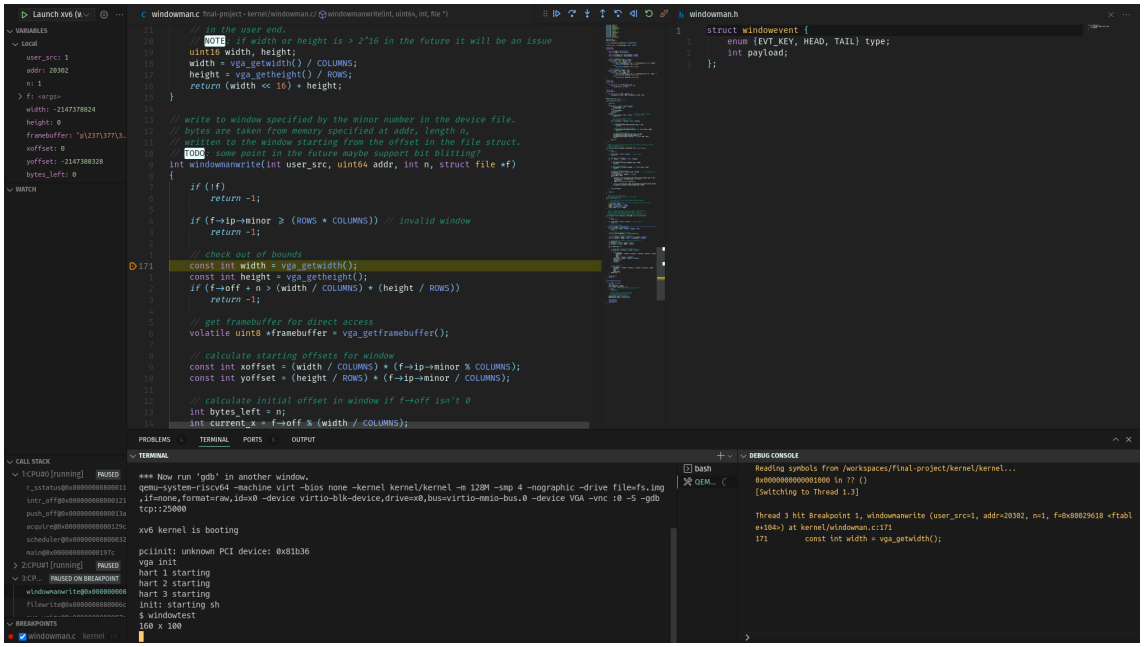

Figure 1.2: Visual Studio Code Debugging

Chapter 2

Methods

2.1 Development plan

2.1.1 Model

The waterfall model will be used following the linear nature of the project. Features in the planned software are linearly dependent, and following this model will allow consistent progress to reach the planned objectives and deliverables.

2.1.2 Version control

As Xv6 is already in a git repository hosted on GitHub [2], we can simply fork the repository for our modifications. Using git will allow the progress of the software implementation to be easily documented.

2.2 Analysis and design

2.2.1 Running Xv6

There are two options of running Xv6. This subsection will explore the pros and cons of both options in this project's use case. The usual choice is to use QEMU, an emulator that can run systems built for the RISC-V architecture.

Pros

- Xv6 built with QEMU in mind
- Extensive debugging capabilities
- Easy graphics card and display emulation
- Easy to install from the Ubuntu software repositories

Cons

- Not very performant

The alternative option is to run it on bare metal. As there aren't many RISC-V processors, the options are limited, with one of the only options being the 'HiFive Unmatched' board by SiFive [12].

Pros

- Dedicated RISC-V processor, very performant
- Ability to observe the solution on real hardware

Cons

- Expensive, costs USD 665
- Would require additional hardware for this project, adding to the already expensive cost of the board
  - VGA compatible graphics card
  - JTAG interface hardware for debugging
- Modifications in Xv6 required as some memory addresses are hardcoded for the emulated QEMU 'virt' board

Given all this, running Xv6 in QEMU is the best option as it will suffice for development and testing of a graphics implementation.

2.2.2 QEMU VGA emulation

There are many emulation options for graphics provided by QEMU [13]. We will choose the most basic VGA card that we can access through PCI communication. This card provides full VGA compatibility which can be configured through its data ports (using the memory mapped PCI port IO addresses), and a linear framebuffer exposed as memory mapped IO which can be mapped anywhere in the PCI MMIO region of the virtual address space.

2.2.3 Solution diagram

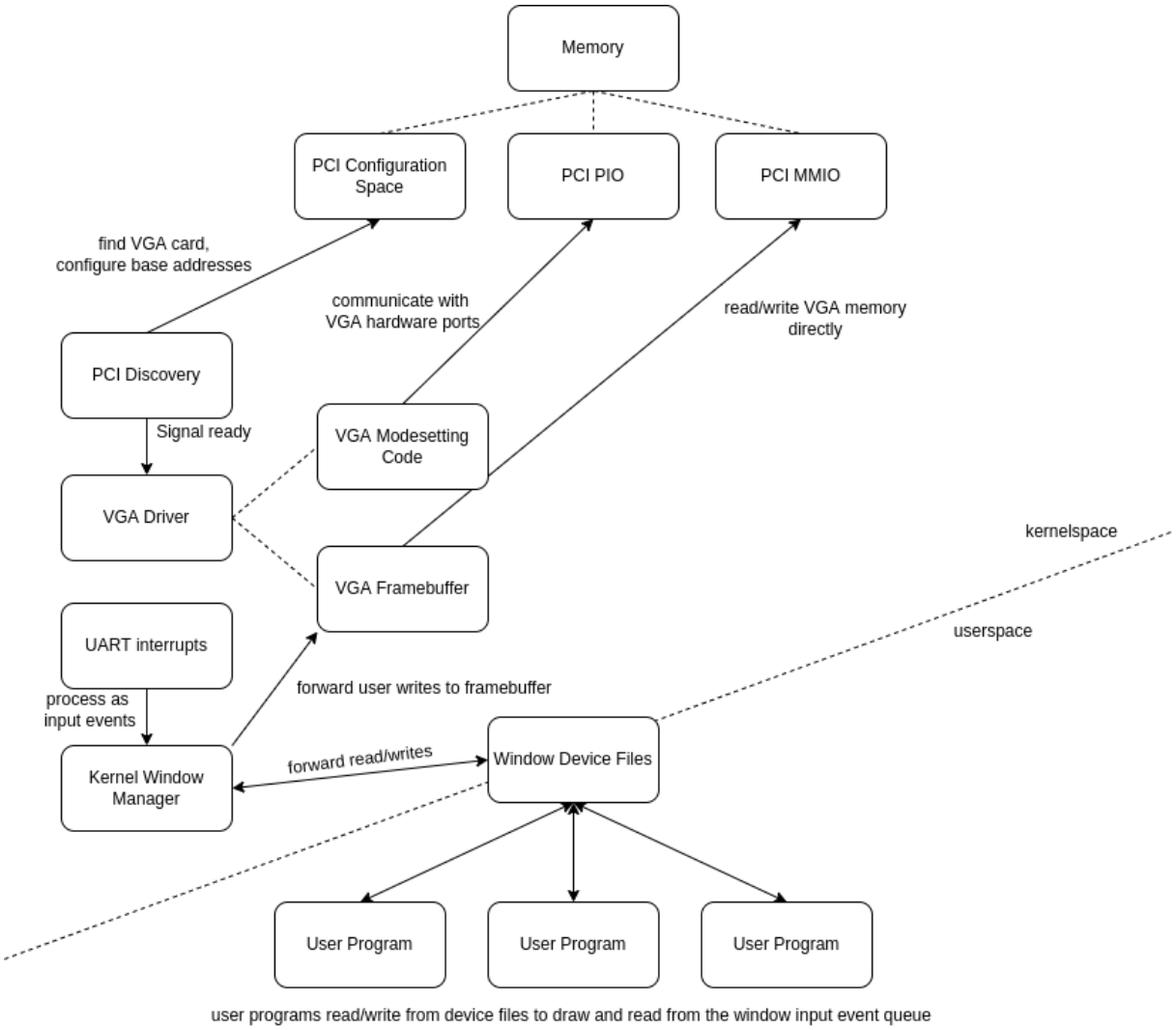

Figure 2.1: Solution design diagram

The design diagram in figure 2.1 illustrates how the different parts of the solution will communicate and interact with each other. In short, the PCI discovery code will find the VGA card and configure the base addresses, then signal the driver when the VGA device is ready.

The VGA driver will provide an interface to the framebuffer and some modesetting code through the port IO and memory IO which have address mappings in the virtual address space. The window manager will then use the framebuffer exposed by the driver to process user code that draws to the screen, and register UART interrupts as input events to be passed back to user code.

2.3 Implementation

2.3.1 Xv6 build modifications

A Makefile is provided with the Xv6 project that contains tasks that builds the kernel, user programs, and the filesystem. It also has tasks that runs the compiled kernel and filesystem

image on QEMU.

QEMU options

There is a variable called `QEMUOPTS` in the Makefile that contains command line options forwarded to QEMU. We need to add the following line

```
QEMUOPTS += -device VGA -vnc :0
```

to enable the VGA device emulation and serve the display output via VNC. We will also modify the C compiler flags to include `-O0` to disable all optimisations during development which could otherwise interfere with debugging.

When creating new files to be compiled into the kernel (Chapters 2.3.3,2.3.4,2.3.5), the appropriate compile targets must be appended to the `OBJS` variable. Similarly for user programs, shared code must be appended to `ULIB`, and the programs appended to `UPROGS`.

2.3.2 Xv6 memory mapping

As mentioned earlier in Chapter 1.2.5, the memory addresses for PCI communication will need to be configured in Xv6. Using the mappings found in the QEMU source tree in Listing 1, we can add the following lines into the kernel pagetable in kernel/vm.c (see Appendix C.2 for address definitions).

```
// map PCI ECAM space
kvmmap(kpgtbl, PCI_ECAM_BASE, PCI_ECAM_BASE, PCI_ECAM_LEN, PTE_R | PTE_W);

// map PCI PIO
kvmmap(kpgtbl, PCI_PIO_BASE, PCI_PIO_BASE, PCI_PIO_LEN, PTE_R | PTE_W);

// map VGA framebuffer (PCI MMIO)
kvmmap(kpgtbl, VGA_FRAMEBUFFER_BASE, VGA_FRAMEBUFFER_BASE,
↪  VGA_FRAMEBUFFER_SIZE, PTE_R | PTE_W);
```

These will configure direct mappings for these addresses for kernel read and write access only, denying execution or unprivileged access. The kernel is now ready to communicate with the emulated PCI VGA device.

2.3.3 PCI communication

VGA device discovery

As shown in figure 1.1, the PCI configuration space is a table of PCI configuration headers where the first 32 bits contain the vendor and device IDs. As documented in the QEMU source tree under /docs/specs/standard-vga.txt [7], the ID for the emulated VGA card is 0x11111234.

With a simple bruteforce scan through the configuration space, through all 256 PCI buses with 32 devices per bus, we can compare the first 32 bits of the header with 0x11111234 (see Appendix C.3).

VGA configuration

Once the device has been found, we need to write data into the configuration header to achieve the following:

- Enable device to respond to port I/O space access, to configure VGA modes and functionality
- Enable device to respond to memory space access, for framebuffer access
- Configure base address in PCI MMIO address space to expose the framebuffer

This can be achieved by the following code snippet, given that 'header' is an unsigned 32 bit pointer to memory.

```
//-- set bits in the command register
// Bit 0 - respond to I/O accesses
// Bit 1 - respond the memory accesses
// Bit 2 - enable bus mastering (allow card to access other components without
↪  CPU intervention)
header[1] = 0b111;

__sync_synchronize(); // ensure order of writes aren't modified by
↪  optimisation

// map the framebuffer in our specified base addr
header[4] = VGA_FRAMEBUFFER_BASE;
```

2.3.4 VGA graphics driver

VGA register access through PCI PIO

VGA hardware is configured through many 8 bit registers, but the hardware only occupies a very short port I/O range [4]. Access is done through indexing, where a register index is written to a port, then data is written/read for the same or adjacent port. On cases where data is fed into the same port as the index, an access operation on a specified "mode setting" port will set the port into index mode.

We can see this functionality in the code snippet in listing 2. The function handles writing to the VGA registers using ports and register indexes, handling the special cases mentioned earlier. The read function is also identical, but instead of writing to 'vga_pio', the contents of the register index at the given data port offset is returned. Note that the order of reads and writes are very important here, so we use `__sync_synchrnize()` to emit a memory fence instruction so the compiler and the CPU do not attempt to change the order of accesses. The `__attibute__((unused))` modifier also informs the compiler that it should not optimise that variable away even though it seems unused, as we need it to perform a read operation on port 0x3DA to set port 0x3C0 into index mode.

```
static volatile uint8 idx_mode_set __attribute__((unused));
static volatile uint8 *vga_pio = (uint8 *)PCI_PIO_BASE;
static void pwrite(uint32 port, uint32 idx, uint8 val)
{
  if (idx == 0xFF)
  {
    // special case, this port doesn't use indices
    // direct write
    vga_pio[port] = val;
    return;
  }

  // perform a read op on 0x3DA to set 0x3C0 into index mode
  idx_mode_set = vga_pio[0x3DA];
  __sync_synchronize();

  // write the index to the port, and the value to port + 1 if not special case 0x3C0
  vga_pio[port] = idx;
  __sync_synchronize();
  uint write_offset = (port == 0x3C0) ? (uint32)0 : (uint32)1;
  vga_pio[port + write_offset] = val;
  __sync_synchronize();

  // perform a read op on 0x3DA to set 0x3C0 into index mode after
  idx_mode_set = vga_pio[0x3DA];
}
```

Listing 2: kernel/pci.c, VGA port write code

Modesetting

VGA hardware modes are configured by setting various registers, to configure the timing and resolution of the output, as well as the colour depth and structure (planar or linear) of the framebuffer. In non text modes, a palette will also need to be configured for the target colour depth.

For the scope of this project, support for only one mode will be implemented, 320x200 linear 256 colour (see Appendix C.4 for modesetting value examples). This mode configures the framebuffer for linear access, with 8 bit (256) colours. This will also make one pixel equal to one byte, simplifying framebuffer manipulation further in addition to linear access.

Setting the mode can be done by writing the appropriate values to the ports (see kernel/vga.h in the project codebase) using the port writing logic written in listing 2.

We will be using the standard VGA 256 colour palette (figure 2.2) for our implementation. The VGA DAC has a colour subsystem that uses an 18 bit colour space, 6 bits for each RGB component. We can store this data as a constant in a 32 bit type array, storing each component

in 8 bits (one byte), using only the first 6 bits in each byte. Writing to the DAC is done in sets of 3, where each component is fed into the DAC data register sequentially (see listing 3).

Figure 2.2: VGA 256 colour palette

```
pwrite(0x3c8, 0xff, 0x00); // set DAC colour register mode to write
for (int i = 0; i < 256; i++)
{
  // take 6 bits from relevant part of palette, shift to uint8 type, and
  ↪ write to reg
  // it is 6 bits because the DAC uses an 18 bit colour space, 6 bits per
  ↪ colour component
  pwrite(0x3c9, 0xff, (VGA_256COL_PALETTE[i] & 0xfc0000) >> 18); // set red
  ↪ val
  pwrite(0x3c9, 0xff, (VGA_256COL_PALETTE[i] & 0x00fc00) >> 10); // set
  ↪ green val
  pwrite(0x3c9, 0xff, (VGA_256COL_PALETTE[i] & 0x0000fc) >> 2);  // set blue
  ↪ val
}
```

Listing 3: kernel/vga.c, setting the VGA palette (see listing 2 for pwrite implementation)

Boot image

During system boot, many operating systems display a boot image while it finishes its boot sequence. Taking inspiration from Linux, the driver will display a boot image, repeated for every CPU in the system. As each CPU core knows its own CPU number, calculating this is trivial, since the width of the screen and the boot image is known (see Appendix C.5).

2.3.5 Grid window manager

Device file handling

As briefly mentioned in Chapter 1.2.6, we will be using device files for programs in userspace to communicate with the window manager. Modifications will need to be made to file IO handling to register our new device file type (major number) and its handlers for read/write operations.

Device files are special files that have a major and minor number. For our case, the major number will indicate that its a 'WINDOW' device file with major number 2, with the minor number specifying which window the file is referring to. There already exists another type of device file 'CONSOLE' in Xv6 with major number 1; this file serves as an abstraction to reading and writing from the UART console. Similarly for the window manager, the new device file will be an abstraction for drawing to a window and reading the window's input events. Using this design pattern saves us from having to add more system calls to Xv6 as well.

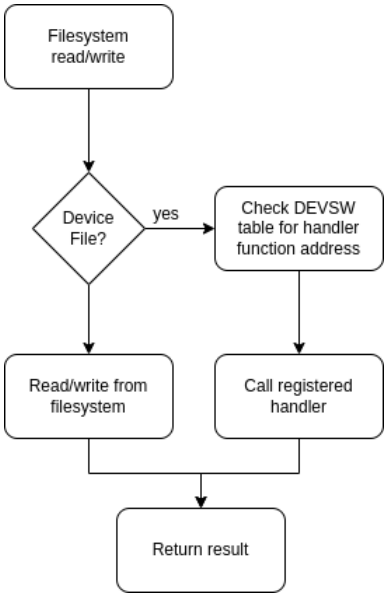

Figure 2.3: Filesystem reads/writes to device files

Registering the function handlers to 'devsw' is very simple: the addresses for the read and write handlers are set on `devsw[MAJOR].read` and `devsw[MAJOR].write` respectively.

Xv6 currently does not pass on the `struct file*` pointer. The function handler signature is `(int, uint64, int)`, but we need details such as the minor number to describe which window and the file offset to determine where in the window we start copying data to. We can simply add the argument for the pointer and modify all the function handlers to also expect that argument.

As each window should only be allocated to one program at a time, we can check the reference count to a device file as it is being opened, and deny access if it already has an existing reference (another program is using the window). Since the grid layout also cannot be changed at runtime in the current implementation, returning the size of the window, the size of the 'file' can also be set during opening. (see listing 4)

Note that the function `windowman_getsize()` applies some bit tricks to fit the width and the height into the uint64 type of the file's size. This is done as the user might find it useful to

```
  if (ip->type == T_DEVICE && ip->major == WINDOW) {
    if (ip->ref > 1) { // only one access per window
      iunlockput(ip);
      end_op();
      return -1;
    }

    ip->size = windowman_getsize();
  }
```

Listing 4: Additional checks when opening a window device file

```
void
uartintr(void)
{
  // read and process incoming characters.
  while(1){
    int c = uartgetc();
    if(c == -1)
      break;
    if (windowmanintr(c) != 0)
      consoleintr(c);
  }

  // send buffered characters.
  acquire(&uart_tx_lock);
  uartstart();
  release(&uart_tx_lock);
}
```

Listing 5: kernel/uart.c, UART interrupt chain

know the dimensions rather than just the number of pixels in the window.

Hooking UART interrupts

Xv6's only source of input is currently UART. Usually human interaction with a system is done through keyboard and mouse input, but Xv6 does not support that and it would be out of scope to implement drivers for those devices. QEMU already forwards keyboard inputs from the host system as UART inputs, so we will be using UART as our input.

Currently the console also uses UART for its input as well, as characters are forwarded to 'consoleintr' as they are received in UART (see listing 5). We can simply add our window manager handler 'windowmanintr' to the chain to also listen to this input. Note that sometimes we want to 'consume' the character (not forward it to other listeners), i.e when a window is active the console should not also receive the input. To achieve this we insert our handler before the console handler, and check its return value to see if it should call the next handler in the chain.

Window switching

A mechanism for cycling between the windows to determine which window is active (if any) is required for multiple reasons. The main use would be to determine which window's event queue normal UART input is appended to, and to provide graphical feedback to the user to indicate the active window (see Appendix C.6 for window border drawing code). We also need a state of no window being active as UART input is also being shared with the console, and when no window is active the input should be forwarded to the console instead.

As there is no mouse input, hotkeys (UART control characters) will be used to cycle between the windows. We can use Control+T to cycle through a static global in the window manager to keep track of which window is active (see Appendix C.6).

Input event queue

Similar to how the console has an input buffer, each window will have an event queue. Each UART character will count as a seperate input. The data structure for a window event is as follows:

```
struct windowevent {
    enum {EVT_KEY, HEAD, TAIL} type;
    uint64 payload;
};
```

The rationale behind having a data structure and a queue is to support other types of events in the future from other human interface devices, such as a mouse. Furthermore, the queue will be implemented as a circular queue, to be more space efficient and reducing the need to shift memory around as events are queued / dequeued. Since the kernel also does not have a dynamic memory allocator, the size of the event queue for each window will also be fixed.

Device read and write functions

When reading from the window device file, the window manager will instead return window events from the event queue (see Chapter 2.3.5) up to the number requested by the read call. The tail of the event queue will also be moved as events are dequeued, to make space for new events (see Appendix C.7 for code implementation).

When writing to the window device file, user data is copied to the framebuffer sequentially, from the file offset specified. Since we are copying user data of a variable length, precautions are needed to defend against potential attacks such as buffer overflows. In addition to that, checks will need to be implemented to also ensure the write call is not overflowing their own allocated window, which can potentially draw over other windows. A single check can address both issues:

```
    // check out of bounds
    const int width = vga_getwidth();
    const int height = vga_getheight();
    if (f->off + n > (width / COLUMNS) * (height / ROWS))
        return -1;
```

After the checks are done, data is copied into the window row by row with `either_copyin()`, an existing function that can take a pointer from a userspace program and copy data from its memory to kernel memory.

```
while (bytes_left > 0)
{
    // check bytes left will overflow this row
    if (bytes_left + current_x >= (width / COLUMNS))
    {
        either_copyin(
            framebuffer + (xoffset + current_x) + (yoffset + current_y) * width,
            user_src,
            addr,
            (width / COLUMNS) - current_x);
        bytes_left -= (width / COLUMNS) - current_x;
        current_x = 0;
        current_y++;
        continue;
    }

    // fits in this row
    either_copyin(
        framebuffer + (xoffset + current_x) + (yoffset + current_y) * width,
        user_src,
        addr,
        bytes_left);
    bytes_left = 0;
}
```

2.3.6 User mode graphics library

The window manager only offers a very raw and crude way to draw to the window. Drawing can only be done on contiguous linear blocks of pixels at a variable offset. To address this, we can write a graphics library in usermode to provide some abstractions such as text rendering, drawing rectangles, rectangular sprites, etc.

Creating an abstraction layer that is designed around the functionality of a window will also be useful for future purposes if the underlying implementation details were to change.

Automatic window allocation and destruction

Since a window can only be accessed at one program at a time, the process of finding a free window can be tedious and will most likely be repeated for all user programs.

We can abstract this logic into a `window_create()` function that returns a handle to the window. The function will find all device files in the root directory that starts with 'window'. If one opens successfully, a data structure defined internally in the library is created containing

the file descriptor and the dimensions, and returned to the user in the form of a opaque pointer (pointer to forward declared struct). We don't want users of the library to directly access this structure in case the implementation changes, therefore we expose it as an opqaue pointer for internal use only.

A complementary `window_destroy()` is also written that clears the window, closes the underlying file descriptor, and frees the handle (struct pointer).

This structure takes inspiration from the 'CreateWindow()' and 'DestroyWindow()' Windows API pattern [9].

Event polling

Our window manager makes event polling very simple: just perform a read syscall with the window device file descriptor with the desired maximum events to read. However since the provided window handle is an opaque type, the library will provide a `window_pollevent()` with a function signature identical to the read syscall, with the exception of the first argument taking the window handle instead of an integer descriptor. The function will forward the arguments to the read systemcall and return the number of events read.

Using this method not only conforms to the design pattern of the library, but has the added benefit of clearer self-documenting code instead of an ambiguous read call to the window descriptor.

Text rendering

The 8x16 font is stored in kernel/font.h where each glyph is encoded in rows where each row is a byte and a bit value of 1 indicates foreground and 0 indicates background.

As shown in listing 6, character rendering is actually quite trivial. Given a character 'c', we get the offset in the ASCII font, and then decode the glyphs. The mask is applied for every pixel as each 8 bits is actually an encoded row. The foreground pixels are then written to the screen with the specified colour and coordinate offset.

Note that this implementation is potentially slow since worst case it may result in 8 * 16 * 2 system calls.

Shape rendering

The ability to draw shape primitives is essential to creating a basic graphical interface. We will implement functionality to draw rectangles, circles and lines.

Drawing rectangles is simple: we simply use `seek()` to draw the rectangle row by row, ensuring we don't overflow the current row.

```
uint8 *rowbuf = malloc(sizeof(uint8) * width);
for (int j = 0; j < height; ++j)
{
    // copy color into rowbuf and write
    memset(rowbuf, color, width);
    seek(win->fd, SEEK_SET, ((y + j) * win->width) + x);
```

```
void window_drawchar(window_handle win, uint x, uint y, char c, uint8 color)
{
    int mask[8] = {128, 64, 32, 16, 8, 4, 2, 1};
    uint8 *glyph = VGA_8x16_FONT + (int)c * 16;

    for (int j = 0; j < 16; j++)
    {
        for (int i = 0; i < 8; i++)
        {
            if (x + i >= win->width || y + j >= win->height)
                continue;

            if (glyph[j] & mask[i])
            {
                seek(win->fd, SEEK_SET, (y + j) * win->width + x + i);
                write(win->fd, &color, 1);
            }
        }
    }

    // reset cursor
    seek(win->fd, SEEK_SET, 0);
}
```

Listing 6: user/uwindow.c, character rendering logic

```
        write(win->fd, rowbuf, width);
    }
```

Drawing circles is also quite simple, although not as performant. Each pixel in the square bounding box of the circle is checked against the distance from the specified centre coodinate, and drawn if within the circle radius.

```
    uint rsquared = r * r;
    for (uint j = y - r; j < y + r && j < win->height; ++j)
        for (uint i = x - r; i < x + r && i < win->width; ++i)
        {
            uint dx = x - i;
            uint dy = y - j;
            uint distsquared = dx * dx + dy * dy;

            if (distsquared <= rsquared)
            {
                seek(win->fd, SEEK_SET, j * win->width + i);
                write(win->fd, &color, 1);
            }
        }
```

Drawing lines is a slightly more complicated process, involving interpolating pixels between

the two specified points. This requires floating point operations as the interval between steps is fractional. It is also not very performant as we draw each pixel individually.

```
int steps = (abs(y1 - y2) + abs(x1 - x2));
float interval = (float)1 / steps;

float t = 0;
for (int i = 0; i < steps; i++, t += interval)
{
    // this is really slow, kernel doesn't support masking / blitting
    // yet so if we don't want to overwrite a rectangular region this is
    // the only way for now
    seek(win->fd, SEEK_SET, interpolate(y1, y2, t) * win->width + interpolate(x1, x2, t))
    write(win->fd, &color, 1);
}
```

Sprite rendering

Rendering rectangular sprites is identical to rendering rectangles, but instead of using a solid colour we use the sprite data for the colours.

```
uint8 *rowbuf = malloc(sizeof(uint8) * width);
for (int j = 0; j < height; ++j)
{
    // copy sprite bytes into rowbuf
    memmove(rowbuf, data + (j * w), width);

    // write to screen
    seek(win->fd, SEEK_SET, ((y + j) * win->width) + x);
    write(win->fd, rowbuf, width);
}
```

2.4 Scheduler modifications

Processes that are drawing graphics should be prioritised as to appear more responsive for the user experience. The current scheduler is a round robin scheduler that may make graphical applications appear less responsive if there are a lot of background processes running.

To address this, a simple priority scheduler will be implemented. Each process will have a priority flag, and the new scheduler algorithm will run priority processes first also in a round robin fashion, otherwise it will run the first available process also with round robin.

Small modifications to kernel/proc.h will be required to support priority processes. Specifically, the priority needs to be added to the processes data structure and a process state enum of 'RESERVED' will also be needed.
The new algorithm will be as follows (see Appendix C.8 for code):

1. Is the current process a priority process? Go to 6 if yes.

2. Is the current process a runnable process? Save the process as a reserve if yes.
3. Go to the next process
4. Have we reached our starting point since last run? Set the reserve process (if exists) and go to 6.
5. Go back to 1.
6. Run process.
7. Go to the next process.
8. Go back to 1.

Finally, a new system call will need to be added to set the running process' priority. This is very simple to do, as the new syscall will only need to set the `p->priority` value to the first argument of the call. See Appendix C.8 for the detailed modifications.

2.5 Version control

Illustrated in the commit graph (figure 2.4), proper version control with a defined workflow was used. 'Feature' branches contain work on mostly isolated features, the 'develop' branch contains work from multiple feature branches, and finally the 'master' branch has merges from 'develop' when the build is stable.

Although this workflow is more suited towards projects where features can be developed concurrently with potentially multiple team members, it provides a nice structure to the commit history. Commits that contribute to a feature can be easily grouped, and there is a dedicated branch for stable builds.

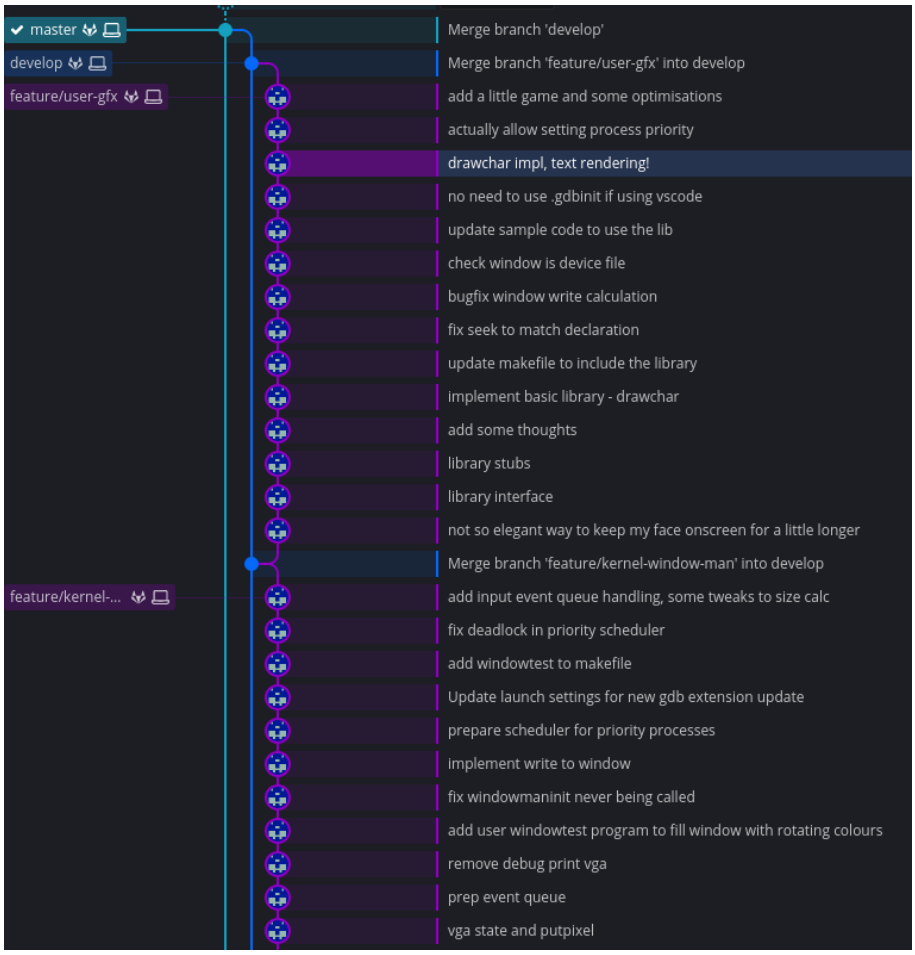

Figure 2.4: Git commit graph

Chapter 3

Results

3.1 Running Xv6 with the solution

A Dockerfile is provided with the project to ensure a consistent compilation and execution environment for development (see Appendix C.1). The file contains instructions for Docker to build a container with the tools required to compile and run Xv6 in QEMU.

Running the solution is simple, create a container from the provided Dockerfile, mount the project directory and run '`make qemu`'. Port 5900 may need to be forwarded from the container to access the VNC output. A configuration for Visual Studio Code also exists in the project, so loading the project and pressing the F5 key should also run the project.

Once QEMU is running, connect a VNC client to localhost:5900 to view the display output.

3.2 Boot sequence test

When the project boots, we should see two things:

- Boot image, one for each CPU in the system (figure 3.1)
- The window borders once the window manager finishes initialisation (figure 3.2)

Figure 3.1: Boot image test

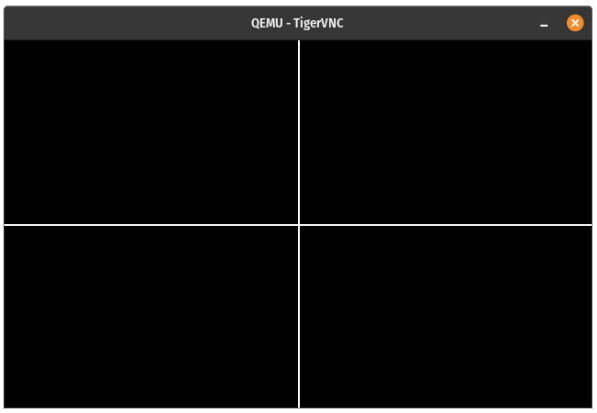

Figure 3.2: Window border drawing test

We can see the boot image works as expected, repeated 4 times offset properly for each CPU core being initialised. The window borders are also being drawn once the system finishes booting.

3.3 Window switching

When pressing the control character specified earlier, we should see the active window being cycled (the red border indicating the active window) and then back to no active window (figure 3.3).

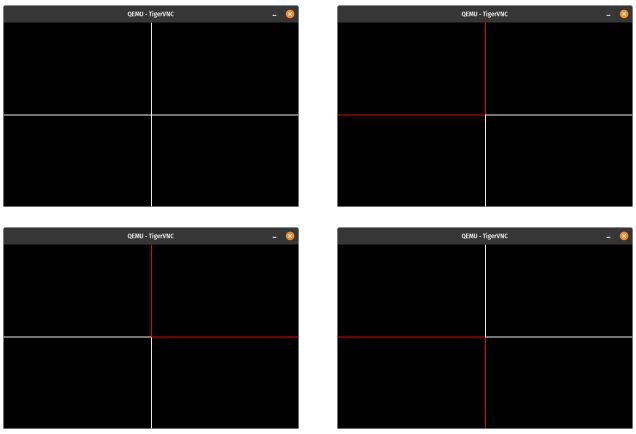

Figure 3.3: Active window cycling test

3.4 Simple display and input test program

Using the graphics library written in Chapter 2.3.6, we can write a simple program to test the event queue and basic drawing.

```
// open the first available window
window_handle win = window_create();

// go through the palette
uint8 cc = 0;
struct windowevent evt;
struct window_dim dim = window_getdimensions(win);
printf("%d x %d\n", dim.width, dim.height);
while (1)
{
    if (window_pollevent(win, &evt, 1) == 1)
    {
        if (evt.payload == 113) // q was inputted
            break;
        cc = cc + 1 % 0xFF;
        window_drawchar(win, 0, 0, ' ', 0x00);
        window_drawchar(win, 0, 0, evt.payload, cc);
        window_drawrect(win, 0, 16, dim.width, dim.height, cc);
    }
}
window_destroy(win);
exit(0);
```

This program draws the character from the event queue and cycles through the colour palette It uses the graphics library to allocate a window, poll the event queue, and draw characters and rectangles to the window.

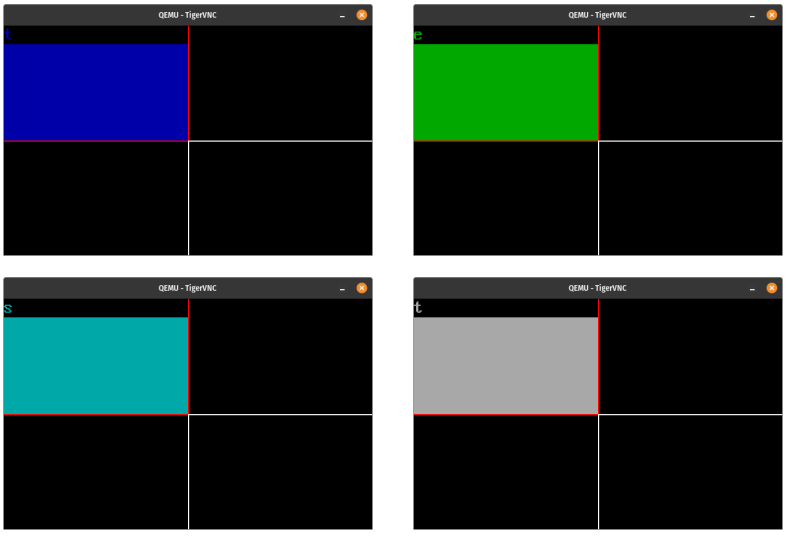

Figure 3.4: user/windowtest program

The screenshots in figure 3.4 show that the window is properly receiving input events and

that the 256 color palette has been loaded.

3.5 High framerate test

We will test the implementation by writing a simple game that clears the screen and draws game sprites multiple times a second (see Appendix C.9 for code).

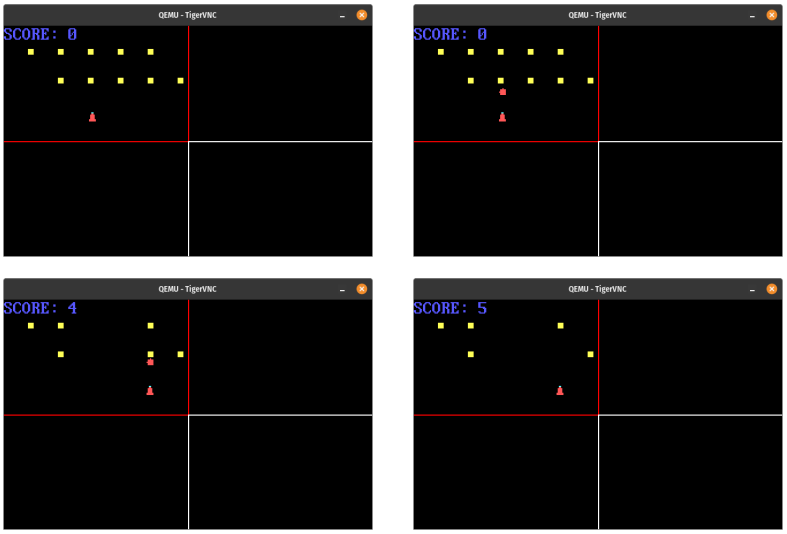

Figure 3.5: user/game program

While running the program, there is a noticeable flickering, and input delay. This is to be expected as our graphics is currently single buffered, so the display will also show the cleared screen between frames. The input delay is probably caused by the numerous system calls when drawing each object to the screen. The game text and the bullets also use `window_drawchar()` and `window_drawcircle()`, which has a slow implementation in which two system calls are made per pixel drawn.

Most if not all of these issues can be resolved by implementing software side double buffering. The graphics library or the kernel will need to store the current frame being written to in memory, and copy it to the main VGA framebuffer when it is ready for display. This will most likely eliminate flickering, and if the buffering is done in userspace, will reduce input lag as the amount of system calls per frame would be reduced to only one.

Chapter 4

Discussion

4.1 Conclusions

The objective of the project, implementing graphics for the Xv6 operating system, has been achieved to a satisfactory level. The solution implementation is able to communicate with a VGA compatible PCI hardware, set it up properly for graphics, and expose the display in the form of a grid of windows to user programs. Furthermore, a graphics library that further abstracts the intricacies of allocating a window and drawing graphics was developed, creating a simple interface for user program development.

4.2 Ideas for future work

Although satisfactory, some aspects of the project can be developed further.

4.2.1 Portability

While the code is mostly portable, some parts were hardcoded to stick to the project timeframe and to stay within the project scope.

Addresses for communication with PCI devices were hardcoded to only support the QEMU 'virt' emulated board. These addresses may be different on other systems, and the proper implementation should be to read the device tree blob placed in memory to retrieve the addresses.

The VGA framebuffer is exposed at a constant location in the PCI MMIO address space. This may cause conflicts if there are other PCI devices also using the PCI MMIO. The PCI code can be improved to keep track of used address spaces and allocate the base addresses dynamically.

The legacy PCI access method is also being used, which may limit discovery of PCI hardware.

4.2.2 Further VGA support

Currently only a limited subset of VGA features are implemented. The driver only supports a single mode, mode 13h, which is a 320x200 resolution 256 colour mode. The implementation could be further improved to implement other modes, or even the extended standards such as SVGA for larger resolutions and a larger colour space.

The VGA driver is also very primitive, it only supports copying memory into the framebuffer. 'Bit blitting' operations could be implemented in the future, where the source data and the framebuffer is combined through bitwise operations. This could greatly optimise draw operations for non rectangular shapes, as the buffer could still be rectangular but pixels can be left unaffected.

4.2.3 Human interfacing

Although slightly out of scope, more human interfacing options could be provided. This could be done by adding USB support for mice, keyboard, gamepads, etc. A mouse cursor could also be implemented in the future.

4.2.4 Advanced graphics library

The current implementation for the graphics library is not ideal for high framerate applications. Double buffering could be implemented to only update the real window framebuffer when it is ready to be displayed, reducing flickering or other graphical artifacts and anomalies.

More utilities could also be included, such as drawing triangle primitives and maybe 2D space transformations. This was planned to be included in the project, but unfortunately there were some issues in QEMU with floating point operations, resulting in the functionality being removed.

Bibliography

[1] MIT. *Xv6 book*. https://pdos.csail.mit.edu/6.S081/2021/xv6/book-riscv-rev2.pdf. 2021.

[2] MIT. *Xv6 source code*. https://github.com/mit-pdos/xv6-riscv. 2021.

[3] IBM. *VGA Technical Reference Manual*. http://www.bitsavers.org/pdf/ibm/pc/cards/IBM_VGA_XGA_Technical_Reference_Manual_May92.pdf. 1992.

[4] *VGA Hardware*. https://wiki.osdev.org/VGA_Hardware. 2020.

[5] J. D. Neal. *VGA Chipset Reference*. http://www.osdever.net/FreeVGA/vga/vga.htm. 1998.

[6] Grant Likely. *Linux and the Devicetree*. https://www.kernel.org/doc/html/latest/devicetree/usage-model.html.

[7] QEMU. *QEMU Source Code - commit e750c10167*. https://github.com/qemu/qemu/tree/e750c10167fa8ad3fcc98236a474c46e52e7c18c. 2021.

[8] *PCI Configuration Space Header*. https://upload.wikimedia.org/wikipedia/commons/c/ca/Pci-config-space.svg.

[9] Microsoft. *Creating a Window*. https://docs.microsoft.com/en-us/windows/win32/learnwin32/creating-a-window. 2021.

[10] Xorg. *X.Org Foundation*. https://www.x.org/wiki/.

[11] Microsoft. *Visual Studio Code*. https://code.visualstudio.com/. 2022.

[12] SiFive. *HiFive Unmatched Datasheet*. https://sifive.cdn.prismic.io/sifive/d0556df9-55c6-47a8-b0f2-4b1521546543_hifive-unmatched-datasheet.pdf.

[13] Gerd Hoffman. *VGA and other display devices in QEMU*. https://www.kraxel.org/blog/2019/09/display-devices-in-qemu/. 2019.

Appendix A

Self-appraisal

A.1 Critical self-evaluation

The development of this project was done well, with all the intended objectives reached (and arguably, more than just the expected outcomes). The code was written in a structured manner and well documented with comments. Proper use of version control was also shown through git, and uploaded on the GitLab platform.

There were a few challenges related to time management, and with respect to that the report writing could have been definitely improved if more time were allocated to iterating through the drafts. In retrospect, the implementation of the graphics library could have definitely been improved if the issues with QEMU floating point instructions were resolved, which wasn't possible partly due to less than ideal time management.

A.2 Personal reflection and lessons learned

This project was definitely a very challenging undertaking. Initial project progress was difficult as documentation for VGA hardware is sparse, especially since it is considered legacy technology. QEMU's board and VGA card emulation also has no format documentation, which resulted in some hours spent looking through QEMU's source code to find things such as addresses and PCI device IDs.

After the initial hurdle of initiating communication with the emulated card and being able to display something on screen, development became easier. This experience also made me realise the importance of documentation and maintainable software. Luckily the QEMU software is well written so it wasn't too difficult to find what I needed in its source code. I had prior experience reading through third party source and technical documentation, and this project challenged those skills.

Overall however, I am happy with what was achieved and I believe the outcomes were more than satisfactory in consideration to the initial objectives of this project. At the start of this project I had no idea of the intricacies of low level communication with graphics hardware, and I managed to develop a fully working graphics implementation with a user API from scratch.

A.3 Legal, social, ethical and professional issues

A.3.1 Legal issues

Xv6 is made open source with the MIT License, which states any person may

> "..use, copy, **modify**, merge, publish, distribute, sublicense, and/or sell copies of the Software..."

free of charge, given that the full license text and copyright notice is provided.

As the software is being modified, to comply with the provided license the original copyright and license will be included in the modified software. Additionally, there are no references to the original authors' rights to intellectual property of Xv6 derivatives, so the project - which are original modifications made to Xv6 - will belong to the author (myself) and should be fine for submission.

A.3.2 Social issues

As there is no processing or storing of personal data there are no social issues.

A.3.3 Ethical issues

There are no ethical issues associated with the project as I am the sole contributor and no personal data was collected or stored.

A.3.4 Professional issues

There are no professional issues associated with the project as I am the sole contributor.

Appendix B

External Material

- Xv6 RISC-V OS
- QEMU
- GNU Compiler Toolchain for RISC-V
- Docker

Appendix C

Project Code Snippets

C.1 Project development environment

```
# Download base image ubuntu 20.04
FROM ubuntu:20.04

# LABEL about the custom image
LABEL maintainer="Raka Gunarto <rakagunarto@gmail.com>"
LABEL version="0.1"
LABEL description="COMP2211 xv6 toolchain"

# Disable Prompt During Packages Installation
ARG DEBIAN_FRONTEND=noninteractive

# Update Ubuntu Software repository
RUN apt update

# Install
RUN apt-get -y install wget git build-essential gdb-multiarch qemu-system-misc
↪  gcc-riscv64-linux-gnu binutils-riscv64-linux-gnu
```

Listing 7: Dockerfile for the development environment

C.2 Xv6 memory layout

```
// in qemu riscv "virt" machine, the address space
// for PCI MMIO (memory mapped IO) addresses are
// 0x40000000 - 0x80000000
//
// realistically we should track the base addresses set for
// pci devices and set them dynamically, but since we're only
// supporting one device it's fine to hardcode the address for now
#define VGA_FRAMEBUFFER_BASE 0x40000000L
#define VGA_FRAMEBUFFER_SIZE 0x1000000L

// in qemu riscv "virt" machine, the configuration space
// starts at 0x30000000 and is 0x10000000 bytes long
#define PCI_ECAM_BASE 0x30000000L
#define PCI_ECAM_LEN 0x10000000L

// in qemu riscv "virt" machine, the port I/O
// address space starts at 0x3000000 and is 0x10000 long
#define PCI_PIO_BASE 0x3000000L
#define PCI_PIO_LEN 0x10000L
```

Listing 8: /kernel/memlayout.h:69-90, PCI and VGA memory addresses

C.3 PCI device discovery

```
void pciinit(void)
{
  uint16 bus; // bus index
  uint8 dev;  // device index

  //-- look through every device (brute-force scan)
  // there are 256 buses, 32 potential devices per bus
  //
  // this is actually very simplified and uses the legacy
  // access method, only looking at PCI Segment Group 0
  // (out of up to 65536 groups)
  for (bus = 0; bus < 256; ++bus)
    for (dev = 0; dev < 32; ++dev)
    {
      //-- calculate offsets
      // volatile pointer because this address is not RAM
      // it's memory mapped PCI registers which can change
      // at any time
      uint32 header_offset = (bus << 16) | (dev << 11);
      volatile uint32 *header = (volatile uint32 *)(PCI_ECAM_BASE +
      ↪  header_offset);

      //-- switch to handle known PCI devices
      // first 32 bits of the PCI header is the
      // device ID and the vendor ID, combined
      // becomes the PCI ID
      switch (*header)
      {
      case 0x11111234:
        setup_vga_card(header);
        break;
      default:
        if (*header != 0xFFFFFFFF)
          printf("pciinit: unknown PCI device: 0x%x\n", header[0]);
        break;
      }
    }
}
```

Listing 9: /kernel/pci.c, PCI device discovery through bruteforce scan (legacy access method)

C.4 VGA modesetting examples

Register name	port	index	mode 3h	mode 12h	mode 13h	mode X
Mode Control	0x3C0	0x10	0x0C	0x01	0x41	0x41
Overscan Register	0x3C0	0x11	0x00	0x00	0x00	0x00
Color Plane Enable	0x3C0	0x12	0x0F	0x0F	0x0F	0x0F
Horizontal Panning	0x3C0	0x13	0x08	0x00	0x00	0x00
Color Select	0x3C0	0x14	0x00	0x00	0x00	0x00
Miscellaneous Output Register	0x3C2	N/A	0x67	0xE3	0x63	0xE3
Clock Mode Register	0x3C4	0x01	0x00	0x01	0x01	0x01
Character select	0x3C4	0x03	0x00	0x00	0x00	0x00
Memory Mode Register	0x3C4	0x04	0x07	0x02	0x0E	0x06
Mode Register	0x3CE	0x05	0x10	0x00	0x40	0x40
Miscellaneous Register	0x3CE	0x06	0x0E	0x05	0x05	0x05
Horizontal Total	0x3D4	0x00	0x5F	0x5F	0x5F	0x5F
Horizontal Display Enable End	0x3D4	0x01	0x4F	0x4F	0x4F	0x4F
Horizontal Blank Start	0x3D4	0x02	0x50	0x50	0x50	0x50
Horizontal Blank End	0x3D4	0x03	0x82	0x82	0x82	0x82
Horizontal Retrace Start	0x3D4	0x04	0x55	0x54	0x54	0x54
Horizontal Retrace End	0x3D4	0x05	0x81	0x80	0x80	0x80
Vertical Total	0x3D4	0x06	0xBF	0x0B	0xBF	0x0D
Overflow Register	0x3D4	0x07	0x1F	0x3E	0x1F	0x3E
Preset row scan	0x3D4	0x08	0x00	0x00	0x00	0x00
Maximum Scan Line	0x3D4	0x09	0x4F	0x40	0x41	0x41
Vertical Retrace Start	0x3D4	0x10	0x9C	0xEA	0x9C	0xEA
Vertical Retrace End	0x3D4	0x11	0x8E	0x8C	0x8E	0xAC
Vertical Display Enable End	0x3D4	0x12	0x8F	0xDF	0x8F	0xDF
Logical Width	0x3D4	0x13	0x28	0x28	0x28	0x28
Underline Location	0x3D4	0x14	0x1F	0x00	0x40	0x00
Vertical Blank Start	0x3D4	0x15	0x96	0xE7	0x96	0xE7
Vertical Blank End	0x3D4	0x16	0xB9	0x04	0xB9	0x06
Mode Control	0x3D4	0x17	0xA3	0xE3	0xA3	0xE3

Modes

3h - 80x25 text mode

12h - 640x480 planar 16 color mode

13h - 320x200 linear 256 color mode

X - 320x240 planar 16 color mode

Figure C.1: VGA register settings for various modes [4]

C.5 Boot image

Figure C.2: Boot image, a picture of the project author (Raka Gunarto) with 8 bit colour

```
static const unsigned int BOOTIMG_WIDTH = 91;
static const unsigned int BOOTIMG_HEIGHT = 96;
static const char BOOTIMG[] = {
    ...
        17,18,234,161,161,161,162,162,163,24,24,24,25,25,25,25,
        25,25,25,25,25,25,25,24,24,24,24,24,162,162,162,162,
        162,162,162,162,162,161,235,234,233,17,0,0,0,0,17,17,
        17,17,17,17,17,17,18,18,18,18,18,
        185,186,186,186,186,186,209,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,17,
        18,234,234,162,162,162,162,162,24,24,24,24,25,25,25,25,
        25,25,25,25,25,25,24,24,24,24,24,24,162,162,162,162,
        162,162,162,162,162,162,161,235,234,18,0,0,0,0,17,17,
        17,17,17,17,17,17,17,17,17,17,18,
        186,186,186,186,186,186,17,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,17,17,18,18,
        18,233,161,162,162,162,162,162,24,24,24,25,25,25,25,25,
        25,25,25,25,25,25,24,24,24,24,24,24,162,162,162,162,
        162,162,163,162,162,162,161,235,235,233,17,0,0,0,0,0,
    ...
}
```

Listing 10: kernel/bootimg.h, code snippet of how the boot image is stored

```
  //-- load the bootimage on screen, offset by CPU
  // find how many images fit in a line on the screen
  static const int xfit = 320 / BOOTIMG_WIDTH;

  // figure out the offset for the image based on the CPU id
  const int xoffset = cpu % xfit;
  const int yoffset = cpu / xfit;

  // copy image to screen framebuffer, taking offset into account
  for (int y = 0; y < BOOTIMG_HEIGHT; y++)
    for (int x = 0; x < BOOTIMG_WIDTH; x++)
      vga_framebuffer[(y * 320) + x + (xoffset * BOOTIMG_WIDTH) + (yoffset * 320 * BOOTIMG_HE
  //--
```

Listing 11: kernel/vga.c, displaying the bootimage per CPU

C.6 Window switching code snippet

```
// uart input interrupts go here
int windowmanintr(int c)
{
    switch (c)
    {
    case C('T'): // control-t switches windows
        if (active_window == ROWS * COLUMNS)
            active_window = 0;
        else
            active_window++;
        drawborders();
        return 0;
        ...
```

Listing 12: kernel/windowman.c, handle control character from UART to cycle windows

```
static void
drawborders()
{
    const int width = vga_getwidth();
    const int height = vga_getheight();

    const int activewin_row = active_window / COLUMNS;
    const int activewin_col = active_window % COLUMNS;

    // draw vertical lines
    const int xinterval = width / COLUMNS;
    for (int col = 1; col < COLUMNS; ++col)
        for (int y = 0; y < height; ++y)
            if ((col == activewin_col || col - 1 == activewin_col) && (y / (height / ROWS)) =
                vga_putpixel(xinterval * col, y, 0x28);
            else
                vga_putpixel(xinterval * col, y, 0x0F);

    // draw horizontal lines
    const int yinterval = height / ROWS;
    for (int row = 1; row < ROWS; ++row)
        for (int x = 0; x < width; ++x)
            if ((row == activewin_row || row - 1 == activewin_row) && (x / (width / COLUMNS))
                vga_putpixel(x, yinterval * row, 0x28);
            else
                vga_putpixel(x, yinterval * row, 0x0F);
}
```

Listing 13: kernel/windowman.c, draw window borders, highlight active window in red

C.7 Window device file functions

```
// reads max n events from winow event queue, with window specified by the minor
// number in the device file.
// copies a windowevent struct or array into addr,
int windowmanread(int user_dst, uint64 addr, int n, struct file *f)
{
    if (!f)
        return -1;

    if (f->ip->minor >= (ROWS * COLUMNS)) // invalid window
        return -1;
```

```
    // find head of event queue
    for (int headidx = 0; headidx < NEVTS; ++headidx)
    {
        // skip if not head
        if (evt_queue[f->ip->minor][headidx].type != HEAD)
            continue;

        // return if queue empty
        if (evt_queue[f->ip->minor][(headidx + 1) % NEVTS].type == TAIL)
            return 0;

        // return all events up to n
        evt_queue[f->ip->minor][headidx].type = EVT_KEY; // set to something
        ↪  else, just not HEAD or TAIL
        int evt_count = 0;
        int current_evtidx = (headidx + 1) % NEVTS;
        uint64 caddr = addr;
        for (;
             evt_count < n && evt_queue[f->ip->minor][current_evtidx].type !=
             ↪  TAIL;
             current_evtidx = (current_evtidx + 1) % NEVTS,
             ++evt_count,
             caddr += sizeof(struct windowevent))
        {
            either_copyout(user_dst, caddr,
            ↪  &evt_queue[f->ip->minor][current_evtidx], sizeof(struct
            ↪  windowevent));
            evt_queue[f->ip->minor][current_evtidx].type = HEAD;
        }

        return evt_count;
    }

    return 0;
}
```

Listing 14: kernel/windowman.c, window device file read function

```
// write to window specified by the minor number in the device file.
// bytes are taken from memory specified at addr, length n,
// written to the window starting from the offset in the file struct.
// TODO: some point in the future maybe support bit blitting?
int windowmanwrite(int user_src, uint64 addr, int n, struct file *f)
{
    if (!f)
        return -1;

    if (f->ip->minor >= (ROWS * COLUMNS)) // invalid window
        return -1;

    // check out of bounds
    const int width = vga_getwidth();
    const int height = vga_getheight();
    if (f->off + n > (width / COLUMNS) * (height / ROWS))
        return -1;

    // get framebuffer for direct access
    volatile uint8 *framebuffer = vga_getframebuffer();

    // calculate starting offsets for window
    const int xoffset = (width / COLUMNS) * (f->ip->minor % COLUMNS);
    const int yoffset = (height / ROWS) * (f->ip->minor / COLUMNS);

    // calculate initial offset in window if f->off isn't 0
    int bytes_left = n;
    int current_x = f->off % (width / COLUMNS);
    int current_y = f->off / (width / COLUMNS);

    while (bytes_left > 0)
    {
        // check bytes left will overflow this row
        if (bytes_left + current_x >= (width / COLUMNS))
        {
            either_copyin(
                framebuffer + (xoffset + current_x) + (yoffset + current_y) *
                ↪  width,
                user_src,
                addr,
                (width / COLUMNS) - current_x);
            bytes_left -= (width / COLUMNS) - current_x;
            current_x = 0;
```

```
            current_y++;
            continue;
        }

        // fits in this row
        either_copyin(
            framebuffer + (xoffset + current_x) + (yoffset + current_y) * width,
            user_src,
            addr,
            bytes_left);
        bytes_left = 0;
    }

    drawborders();
    return 0;
}
```

Listing 15: kernel/windowman.c, window device file write function

C.8 Process priority system call

```
--- a/kernel/syscall.c
+++ b/kernel/syscall.c
@@ -105,6 +105,7 @@ extern uint64 sys_unlink(void);
 extern uint64 sys_wait(void);
 extern uint64 sys_write(void);
 extern uint64 sys_uptime(void);
+extern uint64 sys_setpriority(void);

 static uint64 (*syscalls[])(void) = {
 [SYS_fork]    sys_fork,
@@ -128,6 +129,7 @@ static uint64 (*syscalls[])(void) = {
 [SYS_link]    sys_link,
 [SYS_mkdir]   sys_mkdir,
 [SYS_seek]    sys_seek,
+[SYS_setpriority]    sys_setpriority,
 [SYS_close]   sys_close,
 };

--- a/kernel/syscall.h
+++ b/kernel/syscall.h
@@ -20,4 +20,5 @@
 #define SYS_link   19
```

```
 #define SYS_mkdir  20
 #define SYS_seek  20
-#define SYS_close  21
+#define SYS_setpriority  21
+#define SYS_close  22
--- a/kernel/sysproc.c
+++ b/kernel/sysproc.c
@@ -17,6 +17,18 @@ sys_exit(void)
   return 0;  // not reached
 }

+uint64
+sys_setpriority(void)
+{
+  int prio;
+  if (argint(0, &prio) < 0)
+    return -1;
+  if (prio != 0 || prio != 1)
+    return -1;
+  myproc()->priority = prio;
+  return 0;
+}
+
 uint64
 sys_getpid(void)
 {
--- a/user/user.h
+++ b/user/user.h
@@ -24,6 +24,7 @@ int getpid(void);
 char* sbrk(int);
 int sleep(int);
 int uptime(void);
+int setpriority(int);

 // ulib.c
 int stat(const char*, struct stat*);
diff --git a/user/usys.pl b/user/usys.pl
index
↪  5c787e34826f633b249b63ab338043ea69d64269..2813e432c4dff7703ad5c7203f4cbebb7bbce0e7
↪  100755
--- a/user/usys.pl
+++ b/user/usys.pl
@@ -37,3 +37,4 @@ entry("getpid");
```

```
 entry("sbrk");
```

Listing 16: Diff at commit da05fd8d to add the setpriority() system call

C.9 High framerate game program code

```
const int gunsprite_width = 6;
const int gunsprite_height = 8;
const uint8 gunsprite[] =
{
    0x00, 0x00, 0x0B, 0x0B, 0x00, 0x00,
    0x00, 0x00, 0x0C, 0x0C, 0x00, 0x00,
    0x00, 0x0C, 0x0C, 0x0C, 0x0C, 0x00,
    0x00, 0x0C, 0x0C, 0x0C, 0x0C, 0x00,
    0x00, 0x0C, 0x0C, 0x0C, 0x0C, 0x00,
    0x00, 0x0C, 0x0C, 0x0C, 0x0C, 0x00,
    0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x0C,
    0x0C, 0x0C, 0x0C, 0x0C, 0x0C, 0x0C,
};

struct coords {
    uint x;
    uint y;
};

static void fire(int bulletx, int bullety, struct coords *bullets)
{
    for (int i = 0; i < 10; ++i)
        if (bullets[i].x == -1 && bullets[i].y == -1)
        {
            bullets[i].x = bulletx, bullets[i].y = bullety;
            break;
        }
}

static int colliding(struct coords c1, struct coords c2, int combinedrad)
{
    uint rsquared = combinedrad * combinedrad;
    uint dx = c1.x - c2.x;
    uint dy = c1.y - c2.y;
    uint distsquared = dx * dx + dy * dy;
    if (distsquared <= rsquared)
        return 1;
```

```
    return 0;
}

int main(int argc, char *argv[])
{
    setpriority(1);

    // open the first available window
    window_handle win = window_create();

    struct windowevent evt;
    struct window_dim dim = window_getdimensions(win);

    int gunx = dim.width / 2;
    int guny = (dim.height / 4) * 3;

    const uint bulletrad = 3;
    struct coords bullets[10];

    const uint enemywh = 5;
    struct coords enemies[10];

    for(int i = 0; i < 10; ++i)
    {
        bullets[i].x = -1;
        bullets[i].y = -1;

        int row = i / 5 + 1;
        enemies[i].x = ((i % 5) + row) * (dim.width / 6);
        enemies[i].y = row * (dim.height / 4);
    }

    uint score = 0;
    while (score < 10)
    {
        // clear screen
        window_clearscreen(win);

        // handle inputs
        if (window_pollevent(win, &evt, 1) == 1)
        {
            switch (evt.payload)
            {
```

```
        case 'a':
            gunx = gunx > 0 ? gunx - 1 : gunx;
            break;
        case 'd':
            gunx = gunx < dim.width ? gunx + 1 : gunx;
            break;
        case ' ':
            fire(gunx + gunsprite_width, guny - 2, bullets);
            break;
        default:
            break;
        }
    }

    // movement
    for(int i = 0; i < 10; ++i)
    {
        // bullets, delete when top of screen reached
        if (bullets[i].x != -1 && bullets[i].y != -1)
            if (--bullets[i].y <= 0)
                bullets[i].x = -1, bullets[i].y = -1;


    }

    // collisions
    for(int bullet = 0; bullet < 10; ++bullet)
        if (bullets[bullet].x != -1 && bullets[bullet].y != -1)
            for (int enemy = 0; enemy < 10; ++enemy)
                if (enemies[enemy].x != -1 && enemies[enemy].y != -1 && colliding(bullets
                {
                    score++;
                    bullets[bullet].x = -1, bullets[bullet].y = -1;
                    enemies[enemy].x = -1, enemies[enemy].y = -1;
                    break;
                }

    // draw
    window_drawsprite(win, gunx, guny, gunsprite_width, gunsprite_height, gunsprite);
    for(int i = 0; i < 10; ++i)
    {
        if (bullets[i].x != -1 && bullets[i].y != -1)
            window_drawcircle(win, bullets[i].x - bulletrad, bullets[i].y - bulletrad, bu
```

```
            if (enemies[i].x != -1 && enemies[i].y != -1)
                window_drawrect(win, enemies[i].x - enemywh, enemies[i].y - enemywh, enemywh,
        }

        window_drawchar(win, 8*0, 0, 'S', 0x09);
        window_drawchar(win, 8*1, 0, 'C', 0x09);
        window_drawchar(win, 8*2, 0, 'O', 0x09);
        window_drawchar(win, 8*3, 0, 'R', 0x09);
        window_drawchar(win, 8*4, 0, 'E', 0x09);
        window_drawchar(win, 8*5, 0, ':', 0x09);
        window_drawchar(win, 8*6, 0, ' ', 0x09);
        window_drawchar(win, 8*7, 0, (char)score + 48, 0x09);
    }
    window_destroy(win);
    exit(0);
}
```

Listing 17: user/game.c source code