The JOS Window System

This is a project to add a Window server and GUI programs to the learning operating system JOS. The original implementation was mostly in the kernel, but the final implementation moves most of the code to user mode. Unfortunately, I can't share the code because it is part of an assignment, and the writeup might not make sense if you don't know how JOS works, but there are a few pictures instead.

Hardware initialization

The display adapter is set to graphics mode using VBE. These extensions are a standardized way for the OS to ask for functionality from the VGA adapter. Though recent versions deprecated fixed mode numbers in favor of querying the adapter to find which modes it supports, I shortcutted by asking for a known mode and hardcoding its settings. I chose mode 103h (or 0x103). This is a Super VGA resolution of 800x600, with 1 byte per pixel. Colors are not separated into channels, but instead are indexes in standard color palette.

The mouse is a PS/2 device, so it communicates over IO port 0x60 and 0x64, just like the keyboard. During kernel initialization, I send some commands to activate the mouse so it starts generating interrupts. I had to make sure the kernel only tries to read on those ports when there is an interrupt, and make sure that is knows which one it expects. For the longest time, moving the mouse while pressing a key would break everything. After much debugging, I realized that the hardware sometimes generates keyboard or mouse interrupts before the data is ready. In those cases, the driver should simply ignore it. Every mouse event needs 3 bytes of data, but if the keyboard is also being pressed, the mouse interrupt may fire 4 times, and the first time the keyboard data will be in the input. A bit in the status register shows whether the data is actually for the mouse. Once I figured that out, my system was no longer mouse ^ keyboard, but both could be used at the same time.

I save mouse events in a circular buffer, and added the syscall `sys_get_event`. Every environment has a current index in the buffer, and `sys_get_event` returns the next unread event, if any. If the environment has fallen behind (there are only 1024 events in the buffer), it simply skips to the oldest one available and prints a warning. It is up to the user programs to not react to events when they are not the active window.

Images

256 colors is not a lot, but I like the simplicity of a flat framebuffer and a single byte per pixel. This simplified much of the drawing code, and I like the

aesthetic. I imported the palette into GIMP (a fancy way of saying I used the color picker on each color in the picture above), and then I could export dithered bitmaps. The bitmaps saved in exactly the right format: after the header, every byte was a pixel, and the value of each pixel is exactly what the VGA adapter expects. My bitmap "parser" simply interprets the first part of the file using the bitmapv4 format to find the size of the image and the offset to the pixel data. It does some small sanity checks, but ignores the palette data and assumes the image is saved in indexed mode using the correct palette. I could do more with this, but bitmaps can also support compression including run-length encoding or even JPG/PNG(!). (A bitmap file may just be a header with a bit saying that the image data is actually a JPEG). Implementing all of that would have been more work.

I also have 3 other image formats for specialized data. The mouse pointer image is just an array of integers.

```
#define _ 0
#define X 1
#define O 2
```

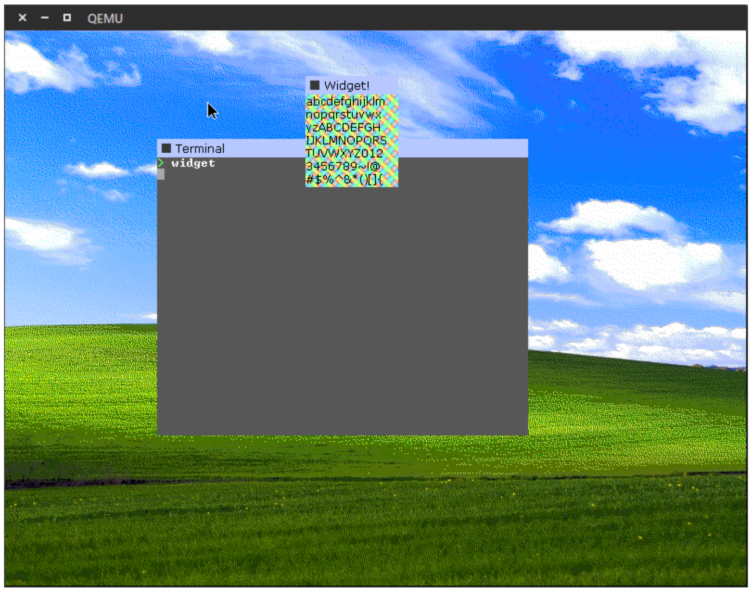
```
static const uint8_t const mouse_pic[] = {
  X,_,_,_,_,_,_,_,_,_,_,_,_,
  X,X,_,_,_,_,_,_,_,_,_,_,_,
  X,O,X,_,_,_,_,_,_,_,_,_,_,
  X,O,O,X,_,_,_,_,_,_,_,_,_,
  X,O,O,O,X,_,_,_,_,_,_,_,_,
  X,O,O,O,O,X,_,_,_,_,_,_,_,
  X,O,O,O,O,O,X,_,_,_,_,_,_,
  X,O,O,O,O,O,O,X,_,_,_,_,_,
  X,O,O,O,O,O,O,O,X,_,_,_,_,
  X,O,O,O,O,O,O,O,O,X,_,_,_,
  X,O,O,O,O,O,O,O,O,O,X,_,_,
  X,O,O,O,O,O,O,O,O,O,O,X,_,
  X,O,O,O,O,O,O,X,X,X,X,X,_,
  X,O,O,O,X,O,O,X,_,_,_,_,_,
  X,O,O,X,_,X,O,O,X,_,_,_,_,
  X,O,X,_,_,X,O,O,X,_,_,_,_,
  X,X,_,_,_,_,X,O,O,X,_,_,_,
  _,_,_,_,_,_,X,O,O,X,_,_,_,
  _,_,_,_,_,_,_,X,X,_,_,_,_
};
#undef _
#undef X
#undef O
```

This is because it must support 2 colors and transparency. I couldn't load files from the disk until lab5, so I didn't implement it this way, but a better way would be to load it from a bitmap and pass a chroma key color to use as transparency.

I have two different fonts as well. I added the fixed width font very early – before I had any windowing, I had a graphics mode terminal emulator to replace the CGA output. Because I didn't even have bitmap support, let alone disk access, I created a Python script which takes fixed-width font images in the format dwarf fortress expects (a 16x16 grid of characters). The dwarf fortress Tileset Repository had a bunch of good fonts. The Python script outputs a C header file which has an array of all 256 characters. Each character is an array of `FONT_HEIGHT` integers, and each integer is a bit field saying which pixels are set. This means it can support up to 32 pixel wide fonts, but the terminal uses an 8x12 font. I could re-implement the Python script in C and have it load as a bitmap file, or change the font-rendering entirely to bitmap drawing, but this worked, so it wasn't a priority.

Proportional fonts came later, but they use a similar format. I created an image file that contains all of the printable ASCII characters in order separated by red lines. A Python script reads the image and creates a bit-field of the characters just as in the fixed-width font, but it also saves the width of each character. I think that I used Tahoma for the font. This could also be ported to C and work on bitmap files.

The Windowing System



```
> widget
```

```
abcdefghijklm
nopqrstuvwx
yzABCDEFGH
IJKLMNOPQRS
TUVWXYZ012
3456789~!@
#$%^&*()[]{
```

Armed with graphics primitives, I built the windowing system. Before I had IPC working, this was implemented in the kernel and used syscalls. However, I was able to move it to user-mode based on the design of JOS's usermode filesystem server.

Like the environments in the kernel, I create an array of Window objects. The windows have location information, an id, an owner, and two buffers. The buffers are allocated on demand (I tried statically allocating all the buffers at first, but it was more RAM than the OS could technically support). Just like the environments, the window's ID mod 0x100 is its index in the window list. When a window is destroyed, its ID increments by 0x100, that way I don't re-use window IDs.

Windows are stored in doubly-linked lists. Redrawing portions of the screen moves from the bottom window to the top window and redraws the part of the window that overlaps with the dirty region. The windows and the screen are double-buffered. The buffer given to client programs is not the buffer used to redraw the screen. The client must call `WINDOW_REDRAW` to copy the contents of (a part of) the buffer to the actual window buffer. I also composite all the windows in a temporary buffer before copying them to the screen. Clicking on a window pulls it to the front of the list.

I wanted to implement fancy things like transparent windows. If I had transparency in windows, I could make the mouse pointer a window as well, and make it always stay on top, just like the desktop background which is a window which always stays on the bottom. I could also add drop-shadows and other effects on the edge of the windows. The VGA palette has 9 colors which are pure black. Some of those could be re-purposed as transparent, or I could provide a chroma key. Performance was the primary reason I didn't – memcpy uses fast x86 instructions that can copy page-by-page instead of byte-by-byte. Doing a conditional copy depending on whether the pixel is transparent would require byte-by-byte copying or SIMD instructions and is unbearably slow. A solution would be to require windows to opt into special rendering, but for now the mouse is simply a special case.

The windowserver listens for IPC commands and sometimes sends results back. For example, receiving the buffer for a window requires the client to send an IPC `WINDOW_GETBUFFER` request with the windowid. Then the client must call `ipc_recv` to receive each page in the buffer until it receives the value 0. (Each buffer is large enough to fill the whole screen, meaning it is 118 pages). A misbehaving client could simply not receive the buffer pages and lockup the windowserver. To fix this I could add a timeout on the attempt to send.

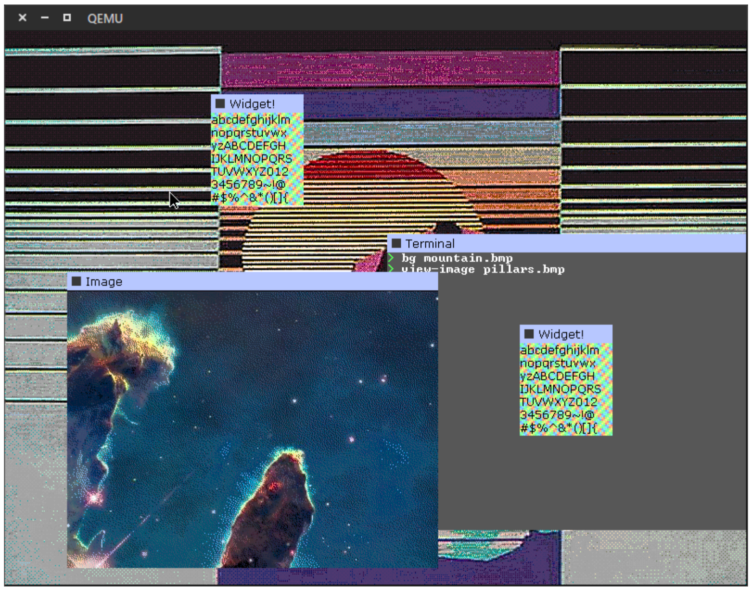
The window server is usually blocked in the `ipc_recv` function, so it cannot poll `sys_get_event`. A simple user mode program does the polling instead, and sends the mouse event to the window server.

```
#include <inc/lib.h>
#include <inc/windowserver.h>

union Windowipc windowipcbuf __attribute__((aligned(PGSIZE)));

void umain(int argc, char **argv) {
  struct InputEvent evt;
  envid_t windowenv = ipc_find_env(ENV_TYPE_GUI);
  while (1) {
    if (sys_get_event(&evt) > 0) {
      windowipcbuf.mouse = evt;
      ipc_send(windowenv, WINDOW_MOUSE, &windowipcbuf, PTE_P | PTE_U | PTE_W
    }
  }
}
```

The `WINDOW_SETBG` command allows you to set the background image by passing a file name in the Windowipc buffer. This is slightly complicated by the fact that the file IO commands use IPC. The responses from the fs server can get mixed up with window commands from other processes. This means that the server must fork and perform the IO in a different environment which is not receiving commands. The forked process can redraw the window buffer, because the buffers are allocated with `PTE_SHARE` set, and once it finishes reading it can ask the server to redraw the whole desktop background window.



```
abcdefghijklm
nopqrstuvwx
yzABCDEFGH
IJKLMNOPQRS
TUVWXYZ012
3456789~!@
#$%^&*()[]{
```

```
> bg mountain.bmp
> view-image pillars.bmp
```

```
abcdefghijklm
nopqrstuvwx
yzABCDEFGH
IJKLMNOPQRS
TUVWXYZ012
3456789~!@
#$%^&*()[]{
```