Design a GUI Interface in the xv6 Operating System

Wenkai Fan
wf39@duke.edu
Duke University
Durham, North Carolina

ABSTRACT

xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix Version 6 (v6). xv6 loosely follows the structure and style of v6, but is implemented for a modern x86-based multiprocessor using ANSI C. In this project, we will extend its capability by adding a simple graphical user interface (GUI) to the system. The GUI should be able to manage multiple program windows, handle user inputs from mouse and keyboard. We are targeting a 800*600 display with 24 bits color.

KEYWORDS

xv6, GUI, user program

ACM Reference Format:
Wenkai Fan. 2020. Design a GUI Interface in the xv6 Operating System. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A GUI (graphical user interface) is a system of interactive visual components for computer software. A GUI displays objects that convey information, and represent actions that can be taken by the user. The objects change color, size, or visibility when the user interacts with them. It is considered to be more user-friendly than a text-based command-line interface, such as MS-DOS, or the shell of Unix-like operating systems. Almost all modern operating systems have their signature GUI interface with different design languages (material design on Android, fluent design on Windows, etc). However, there are many common elements that are widely adopted, like rectangular program window with a title bar, a dock on the desktop to access opened programs, mouse pointer and text cursor that indicate the location for the user interactions. A GUI interface greatly simplifies the user experience and boost productivity in many cases. My goal for this project is to add a GUI interface that is modular (does not affect the existing components of xv6) and contains all the features mentioned above.

Checkout the GitHub page of this project (https://github.com/KevinVan720/xv6-gui) for more information. An introduction video is available at https://warpwire.duke.edu/w/D9AEAA/.

Table 1: Deliverable in this project

Deliverable	Credit	Status
Set up the vesa mode and display something to the screen	10	✓
Mouse support	20	✓
Creating and deleting windows	20	✓
Let windows respond to mouse and keyboard interactions	20	✓
Create some GUI user programs	30	✓

2 BACKGROUND

xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix Version 6 (v6). xv6 loosely follows the structure and style of v6, but is implemented for a modern x86-based multiprocessor using ANSI C. The xv6 provides scheduling, memory management, file system, etc. It can interact with external input devices through interrupts and system calls. After booting the xv6 system, it will spawn the first user process (the shell) on the screen in text mode (organized as 25 lines with 80 characters per line).

Knowing little about OS programming, I started this project with low expectations. I read through the xv6 official book and many other online resources but was still at lost of what to do. Luckily, there are some previous projects about xv6 GUI implementations on GitHub. Many of them were quite old and I could not get them to compile on my machine. There was this one repo (Themis_GUI) that I managed to run. So I started to learn how the author was able to get the GUI part working, at least printing some pixels on the screen. I found that you need to modify the bootloader with some assembly code first to set up the appropriate VGA mode (set to 800*600 resolution with 24 bits color in this project). Then you can print to the screen at individual pixels. Getting this part to work already took me more than a week as the newest xv6 code modifies the starting memory address for external devices. I was just getting page fault errors and a blank screen before I noticed this.

3 ARCHITECTURE OF THE GUI

After setting up the VGA mode, the next step is to develop the GUI part. Again, I borrowed some definitions of struct and primitive painting functions from Themis_GUI. However the architecture is different. Instead of letting every window repaint all the windows above it when it updates, I assign the task of updating the screen only to the desktop process which is now the first process created by "init". So every window only needs to care about handling mouse and keyboard messages sent to itself and updating its own RGB buffer. The desktop process will repaint every window and the mouse on a screen buffer, then flushes the screen buffer to the screen

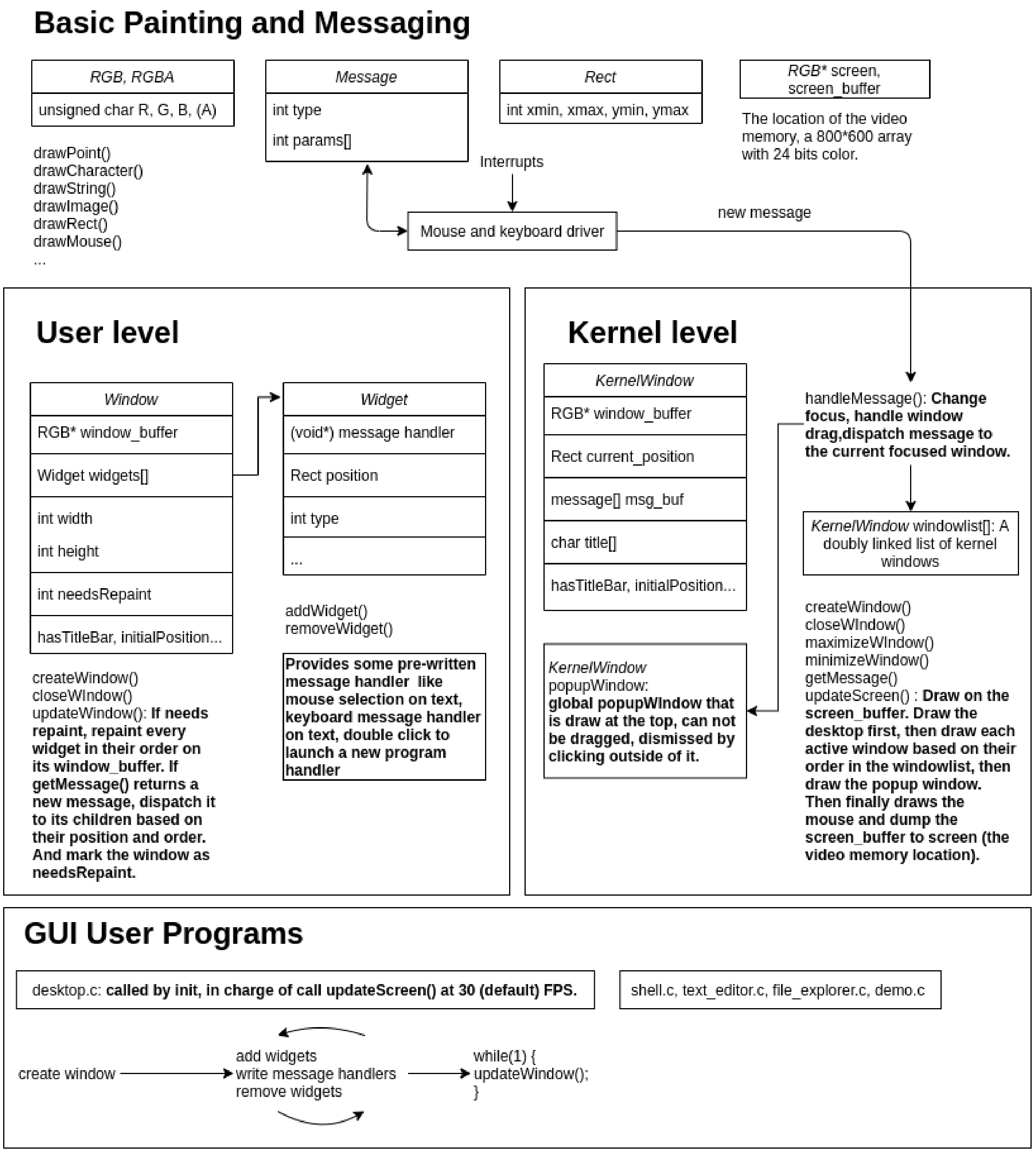

Basic Painting and Messaging
RGB, RGBA
unsigned char R, G, B, (A)
drawPoint()
drawCharacter()
drawString()
drawImage()
drawRect()
drawMouse()
...
Message
int type
int params[]
Rect
int xmin, xmax, ymin, ymax
RGB* screen, screen_buffer
The location of the video memory, a 800*600 array with 24 bits color.
Interrupts
Mouse and keyboard driver
new message
User level
Window
RGB* window_buffer
Widget widgets[]
int width
int height
int needsRepaint
hasTitleBar, initialPosition...
createWindow()
closeWIndow()
updateWindow(): If needs repaint, repaint every widget in their order on its window_buffer. If getMessage() returns a new message, dispatch it to its children based on their position and order. And mark the window as needsRepaint.
Widget
(void*) message handler
Rect position
int type
...
addWidget()
removeWidget()
Provides some pre-written message handler like mouse selection on text, keyboard message handler on text, double click to launch a new program handler
Kernel level
KernelWindow
RGB* window_buffer
Rect current_position
message[] msg_buf
char title[]
hasTitleBar, initialPosition...
KernelWindow popupWindow: global popupWIndow that is draw at the top, can not be dragged, dismissed by clicking outside of it.
handleMessage(): Change focus, handle window drag,dispatch message to the current focused window.
KernelWindow windowlist[]: A doubly linked list of kernel windows
createWindow()
closeWIndow()
maximizeWIndow()
minimizeWindow()
getMessage()
updateScreen() : Draw on the screen_buffer. Draw the desktop first, then draw each active window based on their order in the windowlist, then draw the popup window. Then finally draws the mouse and dump the screen_buffer to screen (the video memory location).
GUI User Programs
desktop.c: called by init, in charge of call updateScreen() at 30 (default) FPS.
shell.c, text_editor.c, file_explorer.c, demo.c
create window
add widgets
write message handlers
remove widgets
while(1) {
updateWindow();
}

Figure 1: Architecture of the GUI

memory at 60 FPS (default). The windows are arranged in a doubly linked list which determines how mouse and keyboard messages are dispatched (from top to bottm) and how windows are painted (from bottom to top).

Every GUI program will manage a window, and adds functionalities to it by adding widgets. The widgets are of types like BUTTON (solid rectangle with centered text), TEXT, INPUTFIELD(text with a text cursor), etc. They are arranged in a doubly linked list which determines how they are painted and receive messages (in a similar fashion like windows). Each widget will be assigned a integer number that can be later used to modify its properties or delete itself (similar to a file descriptor).

The painting and the message handling of those widgets are completely separated. A widget can handle all kinds of messages if the programmer lets it to. So message handling functions can be reused and combined easily. For example, we can write a message handler that handles keyboard input and modifies the INPUTFIELD widget. We can also write a message handler that modifies the text cursor. Then we can combine these two handlers to make a full fledged input text field. Another example is the shell program where the message containing the \n character will be handled differently (start running the typed command instead of change line). But we don't need to write an entirely new message handler, we can just combine this special rule with the existing keyboard handler.

More details of how the GUI part works is in Fig. 1 and also on the GitHub page of this project (https://github.com/KevinVan720/xv6-gui). The code I borrowed from Themis_GUI (gui.c, mouse.c, kbd.c, msg.c) is mainly device driver code and painting primitive functions which is around 1000 lines. The GUI code I wrote is around 2000 lines and the code for setting up five GUI user programs is around 1000 lines.

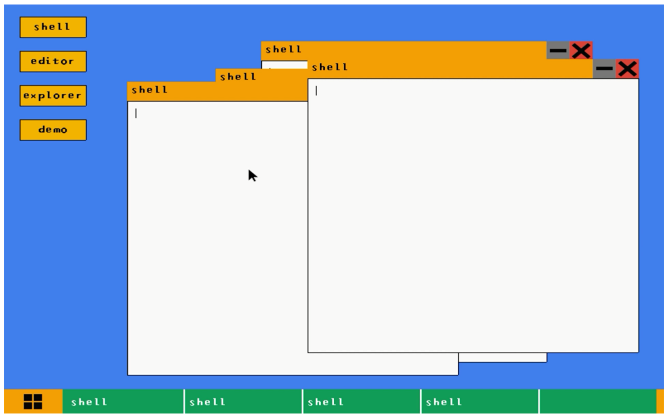
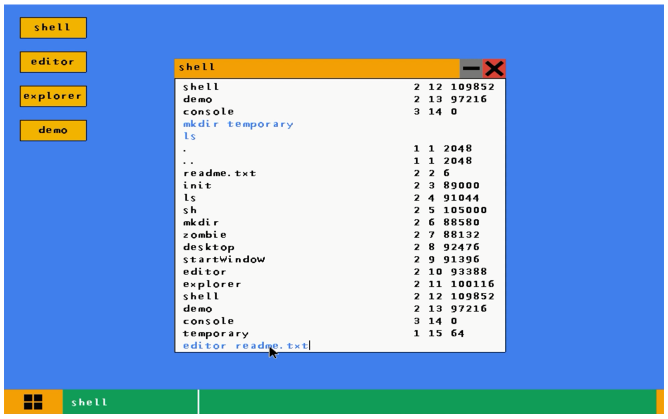


Figure 2: Open multiple program windows.

4 FEATURES OF THE GUI IMPLEMENTATION

In the end, there is a desktop (with a Windows like dock), a shell, a text editor, a file explorer and a demo (flappy bird) program which all work with each other (e.g. you can open a text file in the text editor by using either the shell or the file explorer). Starting each program opens up a new window. All the windows are arranged from bottom to top, can overlap with each other and can be dragged around. Clicking on a window will bring it to the front (focus) and let you interact with it. All the opened windows will also appear on the dock. You can minimize a window, maximize or focus it by



```
shell                                  2 12 109852
demo                                   2 13 97216
console                                3 14 0
mkdir temporary
ls
.                                      1 1 2048
..                                     1 1 2048
readme.txt                             2 2 6
init                                   2 3 89000
ls                                     2 4 91044
sh                                     2 5 105000
mkdir                                  2 6 88580
zombie                                 2 7 88132
desktop                                2 8 92476
startWindow                            2 9 91396
editor                                 2 10 93388
explorer                               2 11 100116
shell                                  2 12 109852
demo                                   2 13 97216
console                                3 14 0
temporary                              1 15 64
editor readme.txt
```

Figure 3: Type commands in the shell program. Window will automatically scroll to the bottom to accommodate long shell history.

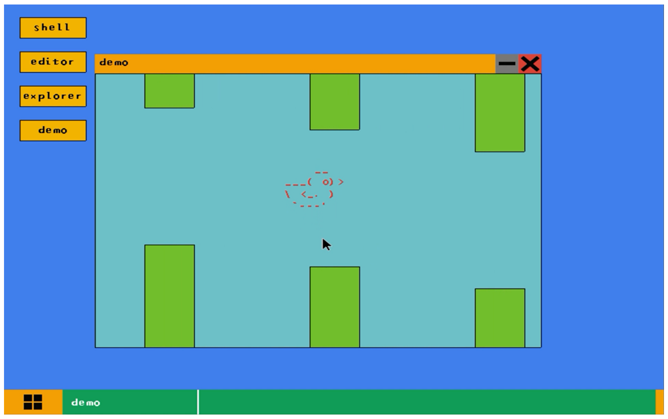


Figure 4: Play flappy bird.

clicking its name on the dock. There is also a minimize all window icon at the bottom right corner. You can click on the Windows icon on the dock to open a popup window. Popup window is always on the top, can be interacted with and will be dismissed once you click outside of it.

For the widgets inside those windows, you can type or delete text, use the mouse or arrow keys to navigate the text cursor in a INPUTFIELD widget. BUTTON widgets usually handles double click mouse messages. For the shell and text editor program, the window will scroll vertically to accommodate long text as you type or move the text cursor. The scrolling is achieved by adding a scrollOffset attribute to the window struct, and widgets can modify this offset when they become too long. This offset is not necessary as we can just directly modify all the relevant widgets' position up and down. However using this single value simplifies the scrolling need of some of the user programs. Because we don't need to manage a list of widget that needs to be scrolled, we just need to modify the scroll offset.

5 CONCLUSION AND OUTLOOK

This project serves as a starting point for more advanced features of a graphical user interface. For example, we can certainly change the linked list structure of widgets into a tree structure, to implement better widget layout mechanisms and inter widget communications. We can add primitive painting functions like painting lines or circles. We can even add system wide copy and paste capabilities through the popup window mechanism. Again, checkout the project's GitHub page for more detailed information.