dUI

C library for making GUIs in the linux TTY

By Daspeller4

Table of Contents

Introduction

Welcome to dUI, a simple library that allows a developer to interface with the linux TTY's framebuffer in an easy and simple way, dUI allows the devloper to place UI elements on the screen, bypassing the need for writing to the framebuffer manually, dUI allows the developer to create interactions for the user by virtue of buttons and the user's access to a mouse cursor. dUI is currently in early development so may lack features, however user feedback is most appreciated and any feature requests/issues should be directed towards the github page.

Getting Started

Installation

Installing dUI is simple, just *`git clone`* the repository and run *`make install`* as root.

```
git clone https://https://github.com/DASPELLER4/dUI
cd dUI
sudo make install
```

Once it has been installed, dUI should be found in `/usr/include`

Creating a program

To include dUI in any C program, include the dUI library

```
#include <dui/dui.h>

int main(){
     return 0;
}
```

However, you will still have the regular TTY screen in front of you, to get a picture on the screen you must initialise the screen, write to it, flush it, and close it.

```
#include <dui/dui.h>

int main(){
     screen_t *screen = createScreen("/dev/fb0");
     renderScreen(screen);
     flushScreen(screen);
     closeScreen(screen);
     return 0;
}
```

First, createScreen opens the /dev/fb0 file for writing and initialises the screen struct for you.

Then, renderScreen takes all of the on-screen text elements, button elements and the mouse and renders it to the screen's draw buffer (where the image is written to before being drawn to the screen.)

Then, flushScreen copies the memory from the draw buffer to the frame buffer and syncs the frame buffer, putting a picture on the screen.

Finally, closeScreen frees all memory and closes the frame buffer.

The screen_t struct contains the dimensions of the screen with the finfo element

```
screen->vinfo.xres // x resolution
screen->finfo.yres // y resolution
```

It also contains the bytes per pixels of the screen, used in most createXElement functions with

```
screen->bpp // bytes per pixel
```

Writing text to the screen

dUI handles text rendering with text elements added to the screen struct itself. To define a text element you use the createTextElement function.

```
text_t* textElement = createTextElement(0, 0, "Example", TEXT_M, WHITE, BLACK, screen->bpp);
```

createTextElement takes in the (x,y) coordinates of the top left pixel of the text element, the contents of the text element, the size of the text, the foreground colour (stored as rgb array), the background colour (also stored as rgb array), and the bytes per pixel.

```
text_t *createTextElement(int x, int y, char *text, int fontSize, uint8_t fg[3], uint8_t bg[3], int bpp);
```

fontSize is the size of the text in pixels divided by 8, where a fontSize of 1 would result in a text element of height 8 pixels and a fontSize of 3 would result in a text element of height 24 pixels.

However, just defining a text element doesn't mean that renderScreen will render it to the draw buffer, in order for it to be included to renderScreen you must add the text to the screen with addText

```
addText(textElement, screen);
```

addText adds a text element to the screen's text element array, so that when renderScreen is next called, the text is rendered to the draw buffer.

Text can be hidden/shown by setting text_t*->visible to false/true (requires stdbool.h).

Putting buttons to the screen

dUI allows you to place buttons on the screen that the user can click on with the mouse to call a function. Functions must be of type void with no arguments.

```
button_t* buttonElement = createButtonElement(0, 0, “Button”, TEXT_M, WHITE, BLACK, myFunction, screen->bpp);
```

createButtonElement takes in the (x,y) coordinates of the top left pixel of the button element, the contents of the text element, the size of the button, the foreground colour (stored as rgb array), the background colour (also stored as rgb array), the callback function that is run on click, and the bytes per pixel.

```
button_t *createButtonElement(int x, int y, char *text, int size, uint8_t fg[3], uint8_t bg[3], void (*onClick)(), int bpp);
```

Size is the size of the button in pixels divided by 10, where a size of 1 would result in a button element of height 10 pixels and a size of 3 would result in a button element of height 30 pixels.

Putting input elements to the screen

Input elements allow users to input text to dUI, they can call a function on key press and on enter press, these functions must also be of type void with no arguments.

```
input_t* inputElement = createInputElement(0, 0, 10, TEXT_M, LT_GREEN, DK_GREEN, screen->bpp);
```

To add callback functions you set the onReturn and onKeyPress struct elements to your functions.

createInputElement takes in the (x,y) coordinates of the top left pixel of the input element, the fixed width of the input element (text that exceeds this gets cut off), the font size, the foreground colour (stored as rgb array), background colour (also stored as rgb array) and the bytes per pixel.

```
input_t *createInputElement(int x, int y, int width, int fontSize, uint8_t fg[3], uint8_t bg[3], int bpp);
```

The text written into the element is available with the input element of the struct. You can access the text input as a zero terminated string with

```
textElement->input
```

Example Program

```
#include <dui/dui.h>
#include <stdio.h>

int keepRunning = 1;
int counter = 0;

void stopButtonFunction(){
	keepRunning = 0;
}

void counterButtonFunction(){
	counter++;
}

int main(int argc, char **argv){
	screen_t *screen = createScreen(“/dev/fb0”);

	text_t *welcomeText = createTextElement(0, 0, “Welcome”, TEXT_L, LT_RED, DK_RED,
screen->bpp);

	int quitXPos = welcomeText->byteWidth/welcomeText->bpp;
	button_t *quitButton = createButtonElement(quitXPos, 0, “Quit”, 2, LT_GREEN, DK_GREEN,
stopButtonFunction, screen->bpp);

	button_t *counterButton = createButtonElement(0, 32, “Count”, 3, LT_BLUE, DK_BLUE,
counterButtonFunction, screen->bpp);

	addText(welcomeText, screen);
	addButton(quitButton, screen);
	addButton(counterButton, screen);

	while(keepRunning){
		renderScreen(screen);
		flushScreen(screen);
	}

	closeScreen(screen);

	printf(“%d\n”, counter);
	return 0;
}
```

The above program defines a text element that says “WELCOME”, and two buttons, quit and count. Once the quit button is pressed, the keepRunning variable is set to 0, the while loop ends and the screen is closed (closing a screen after running addText or addButton frees the text and buttons) and the current count is printed out.

welcomeText->byteWidth is the width of the text element in bytes, therefore if you divide it by bpp you will get the width in pixels.

Dynamic Elements

Changing Text Element

Text Elements can have their size and content changed during runtime but need regeneration. To change the size of a text element you first change the fontSize then call regenerateTextBuffer.

```
textElement->fontSize = TEXT_L;
regenerateTextBuffer(textElement);
```

To change the text content of a text element, you only need to call setTextText

```
setTextText(textElement, “new text”);
```

Changing Button Element

Button Elements can have their size and content changed during runtime but need regeneration. To change the size of a button element you first change the size then call regenerateButtonBuffer.

```
buttonElement->size = 3;
regenerateButtonBuffer(buttonElement);
```

To change the text content of a button element, you only need to call setButtonText

```
setButtonText(buttonElement, “new text”);
```

Derived Screens

A derived screen is a screen that is generated from another screen, when generated, they inherit the memory mapping of the framebuffer, therefore /dev/fb0 is only opened once but you can have two different screens with different text elements and button elements allowing for “window” functionality.

You can create a derived screen with the function deriveScreen

```
screen_t *subScreen = deriveScreen(screen);
```

Macros

```
TEXT_S - small text size (value: 1)
TEXT_M - medium text size (value: 2)
TEXT_L - large text size (value: 4)

WHITE - white colour (value: FFFFFF)
GREY - gre(a)y colour (value: 808080)
BLACK - black colour (value: 000000)
RED - red colour (value: FF0000)
LT_RED - light red colour (value: FF8080)
DK_RED - dark red colour (value: 800000)
ORANGE - orange colour (value: FF8000)
YELLOW - yellow colour (value: FFFF00)
LT_YELLOW - light yellow colour (value: FFFF80)
DK_YELLOW - dark yellow colour (value: 808000)
GREEN - green colour (value: 00FF00)
LT_GREEN - light green colour (value: 80FF80)
DK_GREEN - dark green colour (value: 008000)
CYAN - cyan colour (value: 00FFFF)
BLUE - blue colour (value: 0000FF)
LT_BLUE - light blue colour (value: 8080FF)
DK_BLUE - dark blue colour (value: 000080)
MAGENTA - magenta colour (value: FF00FF)
```