Spark Session: minilibx

updated: 04/03/2021

Session description:

> Learn the basics of working with miniLibX

This tutorial was written with help from Harm Smits and Jelle van Snik's MiniLibX tutorial.

Topics

1. Window Management
2. Pixel Putting
3. More Pixels
4. Events & Hooks

Window Management

Our first step will be to open up some windows! (30 mins)

1. In the set-up instructions, I gave you some code for your `main.c` that included a call to `mlx_init`. But what does it do and what is its prototype? What does it return? (5 mins)
   This link might help -> prototypes
   > Function: initialises mlx, establishes a connection to the correct graphical system
   > Prototype: `void *mlx_init();`
   > Return: mlx instance
2. Let's try opening a small empty window. (10 mins)
   - What is the prototype for `mlx_new_window` and what does it return?
   - How would you declare and initalize it?
   - Now create a window with a width of **800**, height of **480**, and a title of **"My first window"**.
     > Prototype: `void *mlx_new_window(void *mlx_ptr, int size_x, int size_y, char *title);`
     > Return: window instance pointer
     > Code: `void *mlx_win = mlx_new_window(mlx, 800, 480, "My first window");`
3. What happens if you compile and run the program at this point? Your window should have only popped up for a moment. To make it stay longer, we need to use `mlx_loop`. (15 mins)
   - What does it do and what is its prototype?
   - Once you understand that, add `mlx_loop` to your code.
   - Do you now get a window that stays open? Press `Ctrl-C` to close it when you're done admiring your work.
   - **Important**: `mlx_loop` should be called last in your code. Do you know why?
     > Function: loops over the given mlx pointer

> Prototype: `int mlx_loop (void *mlx_ptr); // return unused`
> If mlx_loop is called before any other function, it won't get there.

Break (5 mins)

Pixel Putting

Time to put something on that empty window. (60 mins)

1. Rather than inefficiently pushing pixels one by one to the window using `mlx_pixel_put` , we should draw our pixels onto an **image** first, then push that image to our window. So we need `mlx_new_image` . (10 mins)
    - What is `mlx_new_image` 's prototype and return?
    - Once you understand that, go ahead and initialise an image with a size of **800 x 480**.
      > Prototype: `void *mlx_new_image(void *mlx_ptr,int width,int height);`
      > Return: image instance pointer
      > Code: `void *img_ptr = mlx_new_image(mlx, 800, 480);`
2. In order to know where we can put our pixels, we need to get the **memory address** of our image. That's where `mlx_get_data_addr` comes in. What arguments does it take and what does it return? (10 mins)
   > Prototype: `char *mlx_get_data_addr(void *img_ptr, int *bits_per_pixel, int *size_line, int *endian);`
   > Return: memory address of image
3. Since the function requires a lot of extra variables, let's keep things neat by using a struct for our image data. (10 mins)
   `typedef struct s_img { void *img_ptr; char *address; int bits_per_pixel; int line_size; int endian; } t_img;`
    - Notice that we shifted the image pointer into the struct. Adjust your initialisation of `mlx_new_image` accordingly.
    - Then call `mlx_get_data_addr` and pass it the appropriate arguments/references.
4. As explained in point #1, `mlx_pixel_put` is rather inefficient, so here's a much faster version to use in your code: (10 mins)

```
void    my_pixel_put(t_img *img, int x, int y, unsigned int colour)
{
    char    *dst;
    int     offset;offset = y * img-&gt;line_size + x * (img-&gt;bits_per_pixel /
dst = img-&gt;address + offset;
*(unsigned int *)dst = colour;}
```

    - What is this function doing? What is `offset` ?

      > The function calculates the address of a pixel by adding its memory offset to the address of the first pixel (img->address here).
      > Offset is necessary because `line_size` returned by `mlx_get_data_addr` is different from

> actual window width due to the bytes not being aligned. Function then colours that pixel.
> explanation of formula for offset (see "An image in memory" section)

5. Now, using your `my_pixel_put` function, put a **white** pixel in the **middle** of your image. (10 mins)
   > Code: `my_pixel_put(&img, 800/2, 480/2, 0xFFFFFF);`

6. Our image is all ready to be shown! Let's look at `mlx_put_image_to_window`. What parameters does it take?
   Add the function to your code and see if your little white dot is showing in your window. (10 mins)
   > Prototype: `int mlx_put_image_to_window(void *mlx_ptr, void *win_ptr, void *img_ptr, int x, int y);`

Break (5 mins)

More Pixels

Let's get fancier. Now we're gonna try drawing *lines*. (25 mins)

1. Draw a single horizontal white line running across the middle of the entire screen. You'll need to call `my_pixel_put` in a loop. (15 mins)
   `// example code answer int x = 0; while (x < screen_width) { my_pixel_put(&img, x, screen_height / 2, 0xFFFFFF); x++; }`
2. Now draw a single vertical white line down the middle of the entire screen. (10 mins)
 You should end up with what looks like a crosshair in your window.
   `// example code answer int y = 0; while (y < screen_height) { my_pixel_put(&img, screen_width / 2, y, 0xFFFFFF); y++; }`

Events & Hooks

Having to do `Ctrl-C` every time is probably getting annoying. Let's learn how to close the window when the 'X' button of your window (not your keyboard) is pressed. (35 mins)

1. Hooks, along with events, are vital to making your program interactive. They allow you to intercept keyboard or mouse events and respond to them. You can think of hooks as functions that get called when an event occurs.
 What is the prototype for `mlx_hook`? *(Hint: you may have to look it up in mlx.h)* (5 mins)
   > Prototype: `int mlx_hook(t_win_list *win, int x_event, int x_mask, int (*funct)(), void *param);`
2. miniLibX uses the event codes and masks set out in the **X11** library. What do event codes and masks do? (5 mins)
   - Here's something that might help you understand: event processing
     > Passing event codes & masks allows you to specify which events you want to be notified of
3. What are the **event codes** and **masks** for key presses, key releases, and the 'X' close button? (10 mins)
   - Here's a really helpful resource: handling mouse and keys
   - **Watch out**: the Linux event code for the 'X' close button is different than on macOS. Whereas Mac users can use the code for "DestroyNotify", Linux (and WSL) users will need the code for "ClientMessage".

> Codes: press = 2, release = 3, X button = 17 (Mac) or 33 (Linux)
> Masks: press = `1L << 0`, release = `1L << 1`, X button = `1L << 17`

4. Write a function that: (10 mins)
    - takes as its argument a **pointer to a struct** containing at least your mlx pointer and window pointer *(either make a new struct or expand your existing one)*;
    - destroys your window and exits your program.
5. Add a call to `mlx_hook` in your main that calls this exiting function when the 'X' button is pressed. (5 mins)
    - Does your window close now when you press the 'X' close button on your window?
      `// example code answer int exiter(t_data *game) { mlx_destroy_window(game->mlx_ptr, game->win); exit(0); } int main() { ... mlx_hook(game.win, 33, 1L << 17, exiter, &game); // replace 33 with 17 for Mac }`

Bonus

Let's get some movement on screen: make your crosshair move in 4 directions!

First, however, let's make our crosshair smaller, because who needs a crosshair that big?

1. Expand your struct to include **at least** the following variables you'll need for your drawing function:
    - object width & height;
    - starting x & y positions (i.e. the coordinates of the leftmost pixel of your crosshair).
2. Make a `draw_crosshair` function that:
    - accepts your data/game struct as its parameter;
    - can render a crosshair of a particular **width** and **height**, instead of only the height/width of the screen;
    - renders that crosshair in the **middle of the screen** *(you'll have to do some math using the object dimensions and starting positions, sorry)*;
    - calls `mlx_put_image_to_window` at the end.
3. Get a **30 x 30** pixel crosshair onto your window. Did it work?
   > Note: there will be different ways of solving this. I've included some example non-optimised solutions on the last page of this document.
   > Example formula for computing starting x of crosshair (i.e. the leftmost pixel of the horizontal line):
   `start_x = (game.screen_width - game.obj.width) / 2`

Now let's hook into keyboard events!

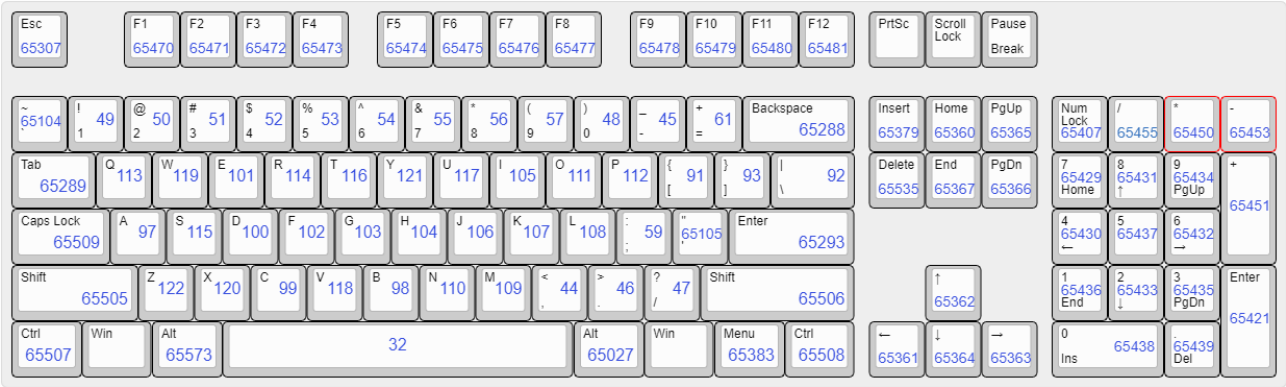
1. Add a call to `mlx_hook` in your main that calls a function `keypress` when keys are…well, pressed.
2. Write that `keypress` function that:
    - calls your exit function when the `ESC` key is pressed;
    - moves the crosshair up, down, left, and right when the corresponding key is pressed.
        - you can choose to use the arrow keys or `W` - `A` - `S` - `D` keys
        - I've included helpful diagrams below for the keycodes you'll need.
3. Add a call to `mlx_loop_hook` in your main that calls a function to render the new image with the modified object coordinates.
4. Do you now have a crosshair that can move across your screen?
    - If you're seeing a trail of crosshairs, you're probably not rendering the background each time.

- If your program is crashing when you hit one of the walls, perhaps you should add checks to your keypress function.

macOS\

\

Linux\



```
// example code for draw_crosshair & keypress functions
void    draw_crosshair(t_data *game)
{
    for (int x = 0; x < game->screen_width; x++)
    {
        for (int y = 0; y < game->screen_height; y++)
            my_pixel_put(game->img, x, y, BLACK); // draw background
    }
    int x_end = game->obj.start.x + game->obj.width; // end of horizontal line
    for (int x = game->obj.start.x; x < x_end; x++)
    {
        int y = game->obj.start.y;
        int y_end = y + 1;
        if (x == x_end - game->obj.width / 2) // for vertical line
        {
            y -= game->obj.height / 2; // top of vertical line
            y_end = y + game->obj.height; // bottom of vertical line
        }
        while (y < y_end)
        {
            my_pixel_put(game->img, x, y, WHITE); // draw crosshair
            y++;
```

```
        }
        mlx_put_image_to_window(game->mlx_ptr, game->win, game->img->img_ptr, 0, 0);
    }

    int     keypress(int keycode, t_data *game)
    {
        if (keycode == ESC)
            exiter(game);
        else if (keycode == MV_UP)
            game->obj.start.y -= 10;
        else if (keycode == MV_DW)
        {
            if (game->obj.start.y + (game->obj.height / 2) + 10
                <= game->screen_height)
                game->obj.start.y += 10;
        }
        else if (keycode == MV_LF)
            game->obj.start.x -= 10;
        else if (keycode == MV_RT)
        {
            if (game->obj.start.x + game->obj.width + 10 <= game->screen_width)
                game->obj.start.x += 10;
        }
        return (0);
    }

    int main(void)
    {
        ...
        game.obj.height = 30, game.obj.width = 30;
        game.obj.start.x = (game.screen_width - game.obj.width) / 2;
        game.obj.start.y = game.screen_height / 2;
        draw_crosshair(&game);

        mlx_hook(game.win, 33, 1L << 17, exiter, &game); // event code on Linux
        mlx_hook(game.win, 2, 1L << 0, keypress, &game);
        mlx_loop_hook(game.mlx_ptr, &updater, &game);
        mlx_loop(game.mlx_ptr);
    }
```