42 Docs

Libs / MiniLibX / Getting started

Getting started

TABLE OF CONTENTS

1 Introduction
2 Installation
   a Compilation on macOS
   b Compilation on Linux
   c Getting a screen on Windows 10 (WSL2)
      a Help! It still does not work!
   d Getting a screen on Windows 11 (WSLg)
3 Initialization
4 Writing pixels to a image
5 Pushing images to a window
6 Test your skills!

Introduction

Now that you know what MiniLibX is capable of doing, we will get started with doing some very basic things. These will provide you with a solid understanding of how to write performant code using this library. For a lot of projects, performance is the essence. It is therefore of utmost importance that you read through this section thoroughly.

Installation

Compilation on macOS

Because MiniLibX requires Cocoa of MacOSX (AppKit) and OpenGL (it doesn't use X11 anymore) we need to link them accordingly. This can cause a complicated compilation process. A basic compilation process looks as follows.

For object files, you could add the following rule to your makefile, assuming that you have the `mlx` source in a directory named `mlx` in the root of your project:

```
%.o: %.c
	$(CC) -Wall -Wextra -Werror -Imlx -c $< -o $@
```

To link with the required internal macOS API:

```
$(NAME): $(OBJ)
	$(CC) $(OBJ) -Lmlx -lmlx -framework OpenGL -framework AppKit -o $(NAME)
```

Do mind that you need the `libmlx.dylib` in the same directory as your build target as it is a dynamic library!

Compilation on Linux

In case of Linux, you can use the Codam provided zip which is a Linux compatible MLX version. It has the exact same functions and shares the same function calls. Do mind, that using memory magic on images can differ as object implementations are architecture specific. Next, you should unzip the MLX for Linux in a new folder, in the root of your project, called `mlx_linux`.

MiniLibX for Linux requires `xorg`, `x11` and `zlib`, therefore you will need to install the following dependencies: `xorg`, `libxext-dev` and `zlib1g-dev`. Installing these dependencies on Ubuntu can be done as follows:

```
sudo apt-get update && sudo apt-get install xorg libxext-dev zlib1g-dev libbsd-dev
```

Now all thats left is to configure MLX, just run the `configure` script in the root of the given repository, and you are good to go.

For object files, you could add the following rule to your makefile, assuming that you have the `mlx` for linux source in a directory named `mlx_linux` in the root of your project:

```
%.o: %.c
	$(CC) -Wall -Wextra -Werror -I/usr/include -Imlx_linux -O3 -c $< -o $@
```

To link with the required internal Linux API:

```
$(NAME): $(OBJ)
	$(CC) $(OBJ) -Lmlx_linux -lmlx_Linux -L/usr/lib -Imlx_linux -lXext -lX11 -lm -lz -o $(NAM
```

Getting a screen on Windows 10 (WSL2)

Getting a screen on WSL2 with Windows 10 can be quite hard. I suggest you either use Windows 11 (as it has graphics support built-in) or use a VM.

Nonetheless, since WSL2 does hot have a graphics layer, you will have to connect to your screen through VNC. This requires a few steps for it to work accordingly:

1. Install Xming, just keep clicking next, the defaults will do. After installing, you will see a little Xming icon in your icon tray. Now exit xming, and open XLaunch, proceed with the following steps:
   - Click `Multiple windows` and go to the next page
   - Click `Start no client` and go to the next page
   - Make sure that the `No Access Control` box is ticked and go to the next page
   - Click `Save configuration` and then `Finish`
2. In WSL execute the following command, this will set your display environment variable accordingly (feel free to create an alias :D):

   ```
   export DISPLAY=$(cat /etc/resolv.conf | grep nameserver | awk '{print $2}'):0.0
   ```

3. Now you can run graphical applications by calling them from your command line interface.

HELP! IT STILL DOES NOT WORK!

Firstly, validate that you are running WSL2. There is a great stackoverflow post about this that will surely help you. If it turns out you are running WSL 1, I strongly encourage you to upgrade, as it's significantly faster. This should also solve your issues. However, if you really want to stick to WSL1, you can also set your display to localhost to solve your issues:

```
export DISPLAY=localhost:0.0
```

After you have validated you are running WSL2, what does the following script output?

```
$> echo $DISPLAY
```

You should be presented an ip address (can be any valid IP address), followed by `:0.0`. If your

output does not contain a real IP address, or is blank, it might be that your `/etc/resolv.conf` file does not exist. Read the contents of the file (using `cat /etc/resolv.conf`) and make sure it looks something like this:

```
# This file was automatically generated by WSL. To stop automatic generation of this file, add
# [network]
# generateResolvConf = false
nameserver 172.27.32.1
```

If the file is empty, try creating a WSL configuration file with the command below and restart your pc for the changes to get applied:

```
echo -e -n "[network]\ngenerateResolvConf = true\n" > /etc/wsl.conf
```

> ***NOTE:*** This requires root proviliges to be executed.

If the file is not empty and contains an IP address, try setting your display directly using the following script. Replace `[YOUR IP]` with the value from your `/etc/resolv.conf` file.

```
export DISPLAY=[YOUR IP]:0.0
```

If I fill in the IP address from my `/etc/resolv.conf` it will look as follows:

```
export DISPLAY=172.27.32.1:0.0
```

If none of that worked, feel free to send me a message on slack (@hsmits).

Getting a screen on Windows 11 (WSLg)

Windows 11's WSL comes with an option to run graphic applications directly. To enable this, follow their official guide for running linux gui apps in wsl. When you have finished the installation, you can simply compile and run minilibx apps and they will appear like an actual application as if they were executed in windows.

Initialization

Before we can do anything with the MiniLibX library we must include the `<mlx.h>` header to access all the functions and we should execute the `mlx_init` function. This will establish a

connection to the correct graphical system and will return a `void *` which holds the location of our current MLX instance. To initialize MiniLibX one could do the following:

```
#include <mlx.h>

int	main(void)
{
	void *mlx;

	mlx = mlx_init();
}
```

When you run the code, you can't help but notice that nothing pops up and that nothing is being rendered. Well, this obviously has something to do with the fact that you are not creating a window yet, so let's try initializing a tiny window which will stay open forever. You can close it by pressing CTRL + C in your terminal. To achieve this, we will simply call the `mlx_new_window` function, which will return a pointer to the window we have just created. We can give the window height, width and a title. We then will have to call `mlx_loop` to initiate the window rendering. Let's create a window with a width of 1920, a height of 1080 and a name of "Hello world!":

```
#include <mlx.h>

int	main(void)
{
	void *mlx;
	void *mlx_win;

	mlx = mlx_init();
	mlx_win = mlx_new_window(mlx, 1920, 1080, "Hello world!");
	mlx_loop(mlx);
}
```

Writing pixels to a image

Now that we have basic window management, we can get started with pushing pixels to the

window. How you decide to get these pixels is up to you, however some optimized ways of doing this will be discussed. First of all, we should take into account that the `mlx_pixel_put` function is very, very slow. This is because it tries to push the pixel instantly to the window (without waiting for the frame to be entirely rendered). Because of this sole reason, we will have to buffer all of our pixels to a image, which we will then push to the window. All of this sounds very complicated, but trust me, its not too bad.

First of all, we should start by understanding what type of image `mlx` requires. If we initiate an image, we will have to pass a few pointers to which it will write a few important variables. The first one is the `bpp`, also called the bits per pixel. As the pixels are basically ints, these usually are 4 bytes, however, this can differ if we are dealing with a small endian (which means we most likely are on a remote display and only have 8 bit colors).

Now we can initialize the image with size 1920×1080 as follows:

```
#include <mlx.h>

int	main(void)
{
	void *img;
	void *mlx;

	mlx = mlx_init();
	img = mlx_new_image(mlx, 1920, 1080);
}
```

That wasn't too bad, was it? Now, we have an image but how exactly do we write pixels to this? For this we need to get the memory address on which we will mutate the bytes accordingly. We retrieve this address as follows:

```
#include <mlx.h>

typedef struct	s_data {
	void *img;
	char *addr;
	int		bits_per_pixel;
	int		line_length;
```

```
	int		endian;
} 			t_data;

int	main(void)
{
	void *mlx;
	t_data	img;

	mlx = mlx_init();
	img.img = mlx_new_image(mlx, 1920, 1080);

	/*
	** After creating an image, we can call `mlx_get_data_addr`, we pass
	** `bits_per_pixel`, `line_length`, and `endian` by reference. These will
	** then be set accordingly for the *current* data address.
	*/
	img.addr = mlx_get_data_addr(img.img, &img.bits_per_pixel, &img.line_length,
	                             &img.endian);
}
```

Notice how we pass the `bits_per_pixel`, `line_length` and `endian` variables by reference? These will be set accordingly by MiniLibX as per described above.

Now we have the image address, but still no pixels. Before we start with this, we must understand that the bytes are not aligned, this means that the `line_length` differs from the actual window width. We therefore should ALWAYS calculate the memory offset using the line length set by `mlx_get_data_addr`.

We can calculate it very easily by using the following formula:

```
int offset = (y * line_length + x * (bits_per_pixel / 8));
```

Now that we know where to write, it becomes very easy to write a function that will mimic the behaviour of `mlx_pixel_put` but will simply be many times faster:

```
typedef struct  s_data {
	void *img;
```

```
	char *addr;
	int		bits_per_pixel;
	int		line_length;
	int		endian;
}			t_data;

void my_mlx_pixel_put(t_data *data, int x, int y, int color)
{
	char *dst;

	dst = data->addr + (y * data->line_length + x * (data->bits_per_pixel / 8));
	*(unsigned int*)dst = color;
}
```

Note that this will cause an issue. Because an image is represented in real time in a window, changing the same image will cause a bunch of screen-tearing when writing to it. You should therefore create two or more images to hold your frames temporarily. You can then write to a temporary image, so that you don't have to write to the currently presented image.

Pushing images to a window

Now that we can finally create our image, we should also push it to the window, so that we can actually see it. This is pretty straight forward, let's take a look at how we can write a red pixel at (5,5) and put it to our window:

```
#include <mlx.h>

typedef struct  s_data {
	void *img;
	char *addr;
	int		bits_per_pixel;
	int		line_length;
	int		endian;
}			t_data;

int  main(void)
{
```

```
	void *mlx;
	void *mlx_win;
	t_data	img;

	mlx = mlx_init();
	mlx_win = mlx_new_window(mlx, 1920, 1080, "Hello world!");
	img.img = mlx_new_image(mlx, 1920, 1080);
	img.addr = mlx_get_data_addr(img.img, &img.bits_per_pixel, &img.line_length,
	                             &img.endian);
	my_mlx_pixel_put(&img, 5, 5, 0x00FF0000);
	mlx_put_image_to_window(mlx, mlx_win, img.img, 0, 0);
	mlx_loop(mlx);
}
```

Note that `0x00FF0000` is the hex representation of `ARGB(0,255,0,0)`.

Test your skills!

Now you that you understand the basics, get comfortable with the library and do some funky stuff! Here are a few ideas:

- Print squares, circles, triangles and hexagons on the screen by writing the pixels accordingly.
- Try adding gradients, making rainbows, and get comfortable with using the rgb colors.
- Try making textures by generating the image in loops.