reactive

42: A Comprehensive Guide to So Long

01/06/2023

Want to provide feedback? Give feedback with Typeform

`so_long` is the first game you will build in your 42 cursus. It's a fun project, as it allows you to use your own graphical assets and animations. It will also significantly help you overcome `cub3d` when you get there, as the MLX is also used.

The main part of your program will be communicating with the MLX and X11, and the rest is a bit of parsing and error handling. We will do a deep dive into the MiniLibX, and then move on to see how we can integrate it within our project. Let's dig in!

What is the MLX?

The MiniLibX, or MLX, is a framework built by Olivier Crouzet on top of X11, a window system developed back in 1984! The MLX is a beginner-friendly C API to interact with the X11 system behind it. Let's look at some of the functions you might be using.

- `mlx_init`: Initialises the MLX library. Must be called before using any other functions.
- `mlx_new_window`: Creates a new window instance.
- `mlx_hook`: Registers events.
- `mlx_loop`: Loops over the MLX pointer, triggering each hook in order of registration.

- **`mlx_xpm_file_to_image`**: Converts an XPM file to an MLX image pointer.
- **`mlx_put_image_to_window`**: Puts your image to the screen at the given coordinates.
- **`mlx_destroy_image`**: Frees the image.
- **`mlx_destroy_window`**: Frees the window instance.
- **`mlx_destroy_display`**: Frees MLX.

We will be looking into each function in more detail later, but if you want more information about these functions, I recommend visiting 42Docs, as they have done a great job documenting the MLX (linked in additional resources).

Installing the MLX

Linking the MLX to your code is rather tricky, so I will try to explain it with both a macOS and Linux machine (works on school computers too). These examples assume you have the MLX installed in your project under a folder called **`mlx`**. If you don't have the MLX installed, you can install it by running this command:

```
git clone https://github.com/42Paris/minilibx-linux.git mlx
```

This will download the MLX in a folder called **`mlx`**, which will help you follow this next part. If you want to ease the installation/build process, you can use **42 CLI**, which has great tooling around the MLX.

MacOS

X11 depends on **`xquartz`**, which you will need to **install from Homebrew** for it to compile successfully.

You will need to include the following headers in your compilation of object files:

```
# Contains the X11 and MLX header files
INCLUDES = -I/opt/X11/include -Imlx

.c.o:
    $(CC) $(CFLAGS) -c -o $@ $< $(INCLUDES)
```

And the following snippet is used for linking the required libraries and frameworks:

```
# Link X11 and MLX, and use OpenGL and AppKit
MLX_FLAGS = -Lmlx -lmlx -L/usr/X11/lib -lXext -lX11 -framework OpenGL -

$(NAME): $(OBJS)
    $(CC) $(CFLAGS) -o $(NAME) $(OBJS) $(MLX_FLAGS)
```

Linux

X11 and MLX depend on several packages, which can all be installed with the below command:

```
sudo apt-get install gcc make xorg libxext-dev libbsd-dev
```

This will install all the required packages (some of which you may already have).

Then you will need to include the required headers in your object files, which looks like this:

```
# Contains the X11 and MLX header files
INCLUDES = -I/usr/include -Imlx

.c.o:
    $(CC) $(CFLAGS) -c -o $@ $< $(INCLUDES)
```

And then link the required libraries:

```
# Link X11 and MLX
MLX_FLAGS = -Lmlx -lmlx -L/usr/lib/X11 -lXext -lX11

$(NAME): $(OBJS)
    $(CC) $(CFLAGS) -o $(NAME) $(OBJS) $(MLX_FLAGS)
```

Universal Compilation

If you develop on both platforms, I suggest making your Makefile work universally between them. This can be accomplished by checking `uname`. Here is a snippet of my Makefile:

```
# [...]

ifeq ($(shell uname), Linux)
    INCLUDES = -I/usr/include -Imlx
else
    INCLUDES = -I/opt/X11/include -Imlx
endif

MLX_DIR = ./mlx
MLX_LIB = $(MLX_DIR)/libmlx_$(UNAME).a
ifeq ($(shell uname), Linux)
    MLX_FLAGS = -Lmlx -lmlx -L/usr/lib/X11 -lXext -lX11
else
    MLX_FLAGS = -Lmlx -lmlx -L/usr/X11/lib -lXext -lX11 -framework Open
endif

# [...]

all: $(MLX_LIB) $(NAME)

.c.o:
    $(CC) $(CFLAGS) -c -o $@ $< $(INCLUDES)

$(NAME): $(OBJS)
    $(CC) $(CFLAGS) -o $(NAME) $(OBJS) $(MLX_FLAGS)
```

```
$(MLX_LIB):
	@make -C $(MLX_DIR)

# [...]
```

> *“Note: Always link your external libs AFTER your object files! Not doing so may lead to an `undefined reference`.”*

Flood Filling the Map

Now that your Makefile is complete, you can move on to parsing. The main part of it is about this rule: “You have to check if there is a valid path in the map”.

This means two things:

1. The player needs access to the exit.
2. The player, prior to exiting the map, is able to collect all collectables.

This can be done in a single recursive function, which follows a similar pattern to the below pseudo-code:

```
if (all_collectables_collected && exit_count == 1)
    return map_valid;
if (on_wall)
    return map_invalid;
if (on_collectable)
    collectables++;
if (on_exit)
    exits++;
replace_current_position_with_wall;
if (one_of_the_four_adjacent_directions_is_possible)
    return map_valid;
return map_invalid;
```

Developing with the MLX

I highly recommend using stack-allocated memory as much as possible, as it will make your life so much easier when you need to exit the program.

I also want to share the data structure I used in this assignment, as I believe it will help many of you to not fall into the same pitfalls I did.

```
typedef struct s_data
{
    void        *mlx_ptr; // MLX pointer
    void        *win_ptr; // MLX window pointer
    void        *textures[5]; // MLX image pointers (on the stack)
    t_map       *map; // Map pointer (contains map details - preferably
}   t_data;
```

Now with that out of the way, let's dig in!

Initialisation

To start working with the MLX, we first need to initialise the MLX library. Under the hood, this creates a new structure which contains all the required data for the MLX to function correctly.

```
#include "mlx/mlx.h"
#include <stdlib.h>

int main(void)
{
    void *mlx_ptr;

    mlx_ptr = mlx_init();
    if (!mlx_ptr)
        return (1);
    free(mlx_ptr);
    return (0);
}
```

In this example, you'll notice we set the `` `mlx_ptr` `` to the result of the `` `mlx_init` `` function. It may return `` `NULL` `` if there was an issue with X11, which you will have to manage.

Creating Your First Window

After instantiating the MLX library, you will be able to create a window. We will be using the `` `mlx_new_window` `` function, which has the following prototype:

```
void *mlx_new_window(void *mlx_ptr, int size_x, int size_y, char *title
```

You'll notice we can provide dimensions, which will come in handy when you load your custom textures. For now though, let's just provide some static variables:

```
#include "mlx/mlx.h"
#include <stdlib.h>

int main(void)
{
    void *mlx_ptr;
    void *win_ptr;

    mlx_ptr = mlx_init();
    if (!mlx_ptr)
        return (1);
    win_ptr = mlx_new_window(mlx_ptr, 600, 400, "hi :)");
    if (!win_ptr)
        return (free(mlx_ptr), 1);
    mlx_destroy_window(mlx_ptr, win_ptr);
    mlx_destroy_display(mlx_ptr);
    free(mlx_ptr);
    return (0);
}
```

This should open a window, and then immediately close it, as we are calling the destroy methods right after. To prevent this behaviour and tell the MLX to wait, we have to learn about hooks and events!

Listening for Events with Hooks

Hooks are an essential part of the MLX, as they allow you to listen for changes and events. Here are a couple of the core hooks that you will be using in **`so_long`**:

- **`mlx_loop`**: Registers any previously defined hooks and listens. This also prevents the default behaviour of destroying the window upon creating it.
- **`mlx_hook`**: Allows you to listen for native X11 events, such as mouse movements, key presses, window interaction, and more... (**full list here**)

With these two hooks, you will be able to listen for user input and mutate data accordingly.

A basic implementation of this could look like this:

```
#include "mlx/mlx.h"
#include <stdio.h>
#include <stdlib.h>
#include <X11/X.h>
#include <X11/keysym.h>

typedef struct s_data
{
    void *mlx_ptr;
    void *win_ptr;
} t_data;

int on_destroy(t_data *data)
{
        mlx_destroy_window(data->mlx_ptr, data->win_ptr);
        mlx_destroy_display(data->mlx_ptr);
        free(data->mlx_ptr);
        exit(0);
        return (0);
    }

    int on_keypress(int keysym, t_data *data)
    {
        (void)data;
        printf("Pressed key: %d\\n", keysym);
        return (0);
    }

    int main(void)
    {
        t_data data;

        data.mlx_ptr = mlx_init();
        if (!data.mlx_ptr)
            return (1);
        data.win_ptr = mlx_new_window(data.mlx_ptr, 600, 400, "hi :)");
        if (!data.win_ptr)
            return (free(data.mlx_ptr), 1);

        // Register key release hook
        mlx_hook(data.win_ptr, KeyRelease, KeyReleaseMask, &on_keypress, &d

        // Register destroy hook
        mlx_hook(data.win_ptr, DestroyNotify, StructureNotifyMask, &on_dest

        // Loop over the MLX pointer
        mlx_loop(data.mlx_ptr);
        return (0);
    }
```

After instantiating MLX and the window, we register two hooks:

- A key release hook, which calls the **`on_keypress`** function every time the user releases a key. This includes normal keys such as **`a`**, **`A`**, and **`1`**, but also some system keys, such as **`ESC`**, **`ENTER`**, and **`DEL`**.
- A destroy notify kook, which calls **`on_destroy`** after the user closes the window.

You'll also notice the last argument of each hook is the argument we want to supply to our hook handler. In this case, we passed a reference to **`data`**, which contains information about our MLX pointers, but you can pass whatever you desire.

> ***"The `KeyRelease` and `DestroyNotify` constants are native X11 events, and can be imported through the X11 headers."***

Loading Textures

Before loading your textures, you may notice you don't have any. Here is a quick way to get some basic graphical assets, and make them ready for MLX:

1. You first need to go to **itch.io**, and choose some assets for your game (you can also make your own in Photoshop, just make sure you export them as PNG).
2. Export each texture into a PNG image.
3. Go to **Convertio**, which will take your images and transform them into an XPM file format.

With these newly created XPM images, I recommend defining both your tile size and the paths for each of your assets in a header file. This will lead to more readable code, and your future self/evaluator will thank you for it.

Loading textures into the MLX is a rather straightforward process. You will be using **`mlx_xpm_file_to_image`** to load your XPM files from your **`/assets`** folder. It has the following prototype:

```
void *mlx_xpm_file_to_image(void *mlx_ptr, char *filename, int *width,
```

You'll want to store the image pointer returned by this function, as you will later need it when you tell the MLX what to render to the screen.

Rendering to your Window

Finally! You can now start rendering textures and *see* something on the screen. For this, there is the MLX function **`mlx_put_image_to_window`**, which, as its name suggests, puts an image on the window. Here is its prototype:

```
int mlx_put_image_to_window(void *mlx_ptr, void *win_ptr, void *img_ptr
```

The image pointer argument is the **`void *`** returned from **`mlx_xpm_file_to_image`**, and **`x`** & **`y`** correspond to the position of the screen from the top left-most pixel of the texture.

Handling Movement

When a player moves, there are two things you need to check:

1. Is the next move valid?
2. Does the next move cause a special event (either winning the game, collecting a collectable, or in the case of a bonus, hitting an enemy)?

Once you outline these two cases, you need to make adjustments to handle these cases separately. Furthermore, you need to make sure the end-user is seeing the changes in real time. This means once the user presses a key, they expect *something*, whether a graphical movement such as the player moving, or a radical change, such as exiting the program after winning or dying.

About Texture Rendering

You will need to make sure you correctly render your assets. You will need to take into consideration layering (i.e. should my player be behind or in front of my floor?), and conditional rendering, such as not rendering the exit if the player doesn't have all collectables. You may also bypass a few of these issues by customising your assets to include the background, which will simplify your code.

Common Mistakes

- Rendering all textures in every frame. This uses up memory and after a (very long) while will cause your program to crash.
- Not using constants to increase readability/maintainability. Although not a bug, it isn't a good habit to explicitly index into an array with a fixed number, use a plain string as an asset path, or repeat an arbitrary value as your tile size.
- Edge cases for map checking. Think of invalid permissions, a map with too few columns/rows, or even an empty line in the middle of the map.
- Going to the exit without all collectables should not end the game.

Conclusion

I hope this helped you complete `` `so_long` ``, and if it did, I'd appreciate you sharing with other 42 students!

Thanks for reading! 🙃

Additional Resources

- 42Docs by Harm Smits
- 42 CLI by Herbie Vine
- Xquartz on Homebrew

- **Full X11 Manual by Christophe Tronche**

Want to provide feedback? Give feedback with Typeform