MiniLibX Python Manual

Nora de Fitero Teijeira (dde-fite)

Abstract

This documentation is a PORT of the ORIGINAL MiniLibX docs. It describes the Python package that provides access to the MiniLibX graphics library. It allows creating windows, drawing pixels, handling images, and capturing keyboard and mouse input through a thin wrapper over the original C API, keeping function names and behavior as close as possible to the native MiniLibX library.

Acknowledgements

The original MiniLibX documentation was created by Olivier Crouzet under the MIT license. This is a derivative work based on his work, created on a non-profit basis with the aim of sharing knowledge among the 42 student community. This derivative work is published under the MIT license by Nora de Fitero Teijeira.

Table of Contents

Introduction

The MiniLibX Python Wrapper allows you to create graphical software easily without any knowledge of X-Window/Wayland/Vulkan on Unix/Linux, or AppKit on macOS. It provides:

- Window creation and management
- Pixel-level drawing
- Image manipulation for faster rendering
- Keyboard and mouse input handling
- PNG and XPM image loading

How a graphics server works

This library interacts with the underlying graphics system of your operating system. Before diving into usage, it's helpful to understand how graphics servers manage windows and handle user input.

Historical X-Window concept

X-Window is a network-oriented graphical system for Unix. It is based on two main parts:

- On one side, your software wants to draw something on the screen and or get keyboard & mouse entries.
- On the other side, the X-Server manages the screen, keyboard and mouse (It is often referred to as a "display").

A network connection must be established between these two entities to send drawing orders (from the software to the X-Server), and keyboard/mouse events (from the X-Server to the software).

Nowadays, most of the time, both run on the same computer.

Modern graphical approach

Modern computers come with a powerful GPU that is directly accessed by applications. Along GPU libraries like Vulkan or OpenGL, the Wayland protocol ensure communication with the compositor program that manages the various windows on screen and the user input events. For your own application:

- The Vulkan or OpenGL library allow you to directly draw any content into your window.

- The Wayland compositor handles the place of your window on screen and send you back the keyboard and mouse inputs from the user.

Unfortunately, this gain of graphical power through GPU access removes the networking aspects that exist with X-Window. It is not possible for a program to access a remote GPU and show its window on a remote display. But current software architectures are more likely based on a local display application that gets data in JSON through a web API.

Getting started

Requirements

Arch Linux

```
sudo pacman -S libxcb xcb-util-keysyms zlib libbsd vulkan-icd-loader
↪  vulkan-tools shaderc
```

Debian/Ubuntu

```
sudo apt install libxcb libxcb-keysyms libvulkan libz libbsd glslc
```

Installation

First compile MiniLibX.

```
make install
```

Create a virtual environment with your preferred manager and open it:

- For bash/zsh:

```
python -m venv .venv
source .venv/bin/activate
```

- For fish:

```
python -m venv .venv
source .venv/bin/activate.fish
```

And install the package:

```
pip install mlx_CLXV-2.2-py3-none-any.whl
```

Example of use

This small Python script displays a small black window with text. It will also print the screen dimensions to stdout and listen for user clicks.

We will explain the functions used in this example later.

```python
from mlx import Mlx

def mymouse(button, x, y, mystuff):
    print(f"Got mouse event! button {button} at {x},{y}.")

def mykey(keynum, mystuff):
    print(f"Got key {keynum}, and got my stuff back:")
    print(mystuff)
    if keynum == 32:
        m.mlx_mouse_hook(win_ptr, None, None)

m = Mlx()
mlx_ptr = m.mlx_init()
win_ptr = m.mlx_new_window(mlx_ptr, 200, 200, "test")
m.mlx_clear_window(mlx_ptr, win_ptr)
m.mlx_string_put(mlx_ptr, win_ptr, 20, 20, 255, "Hello PyMlx!")
(ret, w, h) = m.mlx_get_screen_size(mlx_ptr)
print(f"Got screen size: {w} x {h} .")

stuff = [1, 2]
m.mlx_mouse_hook(win_ptr, mymouse, None)
m.mlx_key_hook(win_ptr, mykey, stuff)

m.mlx_loop(mlx_ptr)
```

Behind the Scenes

When an instance of the Mlx class is created, the first thing it does is construct the path to the C library called libmlx.so.

```python
def __init__(self):
  module_dir = os.path.dirname(os.path.abspath(__file__))
  self.so_file = os.path.join(module_dir, "libmlx.so")
  #...
```

- __file__ is a special Python variable that contains the path to the file where this code is executed.

- os.path.dirname extracts the path from **file** and uses os.path.join to create the path to the library.

It then declares the mlx_func variable, which acts as a bridge between Python and C. Using the CDLL function from Python's ctypes module, it loads the library and calls the original functions.

```
def __init__(self):
  # ...
  self.mlx_func = CDLL(self.so_file)
  # ...
```

For each C function available in the Python wrapper, there is a declaration within the Mlx class:

```
def mlx_init(self):
  self.mlx_func.mlx_init.restype = c_void_p
  return self.mlx_func.mlx_init()
```

You can see how it calls mlx_func.mlx_init(). This mlx_init() is already the original C function. It is necessary to specify the data type returned by the function with mlx_init.restype, which in this case is c_void_p (equivalent to void *).

All of this is passed to CDLL, which, using the previously loaded library, will execute the function and return whatever the function returns.

Initialization and cleanup: mlx_init(), mlx_release()

Synopsis

```
from mlx import Mlx

def mlx_init() -> int: # void *

def mlx_release() -> int: # void *
```

Description

First of all, you need to initialize the connection between your software and the graphic and user sub-systems. Once this completed, you'll be able to use other MiniLibX functions to send and receive the messages from the display, like "I want to draw a yellow pixel in this window" or "did the user hit a key?".

The mlx_init function will create this connection. No parameters are needed, ant it will return a void * identifier, used for further calls to the library routines. The mlx_release function can be used at the end of the program to disconnect from the graphic system and release resources.

If **mlx_init()** fails to set up the connection to the display, it will return None.

Return values

If **mlx_init()** set up the connection to the display correctly, it will return an **int as a pointer**; otherwise, it returns **None**.

Managing windows: mlx_new_window(), mlx_clear_window, mlx_destroy_window

Synopsis

c_void_p, c_uint, c_uint, c_char_p

```
def mlx_new_window(mlx_ptr: int, width: int, height: int, title: str )
↪  -> int: # void *

def mlx_clear_window(mlx_ptr: int, win_ptr: int) -> int:

def mlx_destroy_window(mlx_ptr: int, win_ptr: int ) -> int:
```

Description

The **mlx_new_window** () function creates a new window on the screen, using the *width* and *height* parameters to determine its size, and *title* as the text that should be displayed in the window's title bar. The *mlx_ptr* parameter is the connection identifier returned by **mlx_init** () (see the **mlx** man page). **mlx_new_window** () returns a *void* * window identifier that can be used by other MiniLibX calls. Note that the MiniLibX can handle an arbitrary number of separate windows.

mlx_clear_window () and **mlx_destroy_window** () respectively clear (in black) and destroy the given window. They both have the same parameters: *mlx_ptr* is the screen connection identifier, and *win_ptr* is a window identifier.

Return values

If **mlx_new_window()** fails to create a new window (whatever the reason), it will return NULL, otherwise a non-null pointer is returned as a window identifier. **mlx_clear_window** and **mlx_destroy_window** return nothing.

Drawing inside windows: mlx_pixel_put(), mlx_string_put()

Synopsis

`int`

mlx_pixel_put (*void *mlx_ptr, void *win_ptr, unsigned int x, unsigned int y, unsigned int color*);

`int`

mlx_string_put (*void *mlx_ptr, void *win_ptr, unsigned int x, unsigned int y, unsigned int color, char *string*);

Description

The **mlx_pixel_put** () function draws a defined pixel in the window *win_ptr* using the (*x* , *y*) coordinates, and the specified *color* . The origin (0,0) is the upper left corner of the window, the x and y axis respectively pointing right and down. The connection identifier, *mlx_ptr* , is needed (see the **mlx** man page).

Parameters for **mlx_string_put** () have the same meaning. Instead of a simple pixel, the specified *string* will be displayed at (*x* , *y*).

Both functions will discard any display outside the window. This makes **mlx_pixel_put** slow. Consider using images instead.

Color management

The *color* parameter has an unsigned integer type. The displayed colour needs to be encoded in this integer, following a defined scheme. All displayable colours can be split in 3 basic colours: red, green and blue. Three associated values, in the 0-255 range, represent how much of each colour is mixed up to create the original colour. The fourth byte represent transparency, where 0 is fully transparent and 255 opaque. Theses four values must be set inside the unsigned integer to display the right colour. The bytes of this integer are filled as shown in the picture below:

```
| B | G | R | A |   colour integer
+---+---+---+---+
```

While filling the integer, make sure you avoid endian problems. Example: the "blue" byte will be the least significant byte inside the integer on a little endian machine.

Handle events:

Synopsis

`int`

mlx_loop (*void *mlx_ptr*);

`int`

mlx_key_hook (*void *win_ptr, int (*funct_ptr)(), void *param*);

`int`

mlx_mouse_hook (*void *win_ptr, int (*funct_ptr)(), void *param*);

`int`

mlx_expose_hook (*void *win_ptr, int (*funct_ptr)(), void *param*);

`int`

mlx_loop_hook (*void *mlx_ptr, int (*funct_ptr)(), void *param*);

`int`

mlx_loop_exit (*void *mlx_ptr*);

Events

The graphical system is bi-directional. On one hand, the program sends orders to the screen to display pixels, images, and so on. On the other hand, it can get information from the keyboard and mouse associated to the screen. To do so, the program receives "events" from the keyboard or the mouse.

Description

To receive events, you must use **mlx_loop** (). This function never returns, unless **mlx_loop_exit** is called. It is an infinite loop that waits for an event, and then calls a user-defined function associated with this event. A single parameter is needed, the connection identifier *mlx_ptr* (see the **mlx manual).**

You can assign different functions to the three following events:
- A key is released
- The mouse button is pressed
- A part of the window should be re-drawn (this is called an "expose" event, and it is your program's job to handle it in the Unix/Linux X11 environment, but at the opposite it never happens on Unix/Linux Wayland-Vulkan nor on MacOS).

Each window can define a different function for the same event.

The three functions **mlx_key_hook** (), **mlx_mouse_hook** () and **mlx_expose_hook** () work exactly the same way. *funct_ptr* is a pointer to the function you want to be called when an event occurs. This assignment is specific to the window defined by the *win_ptr* identifier. The *param* address will be passed back to your function every time it is called, and should be used to store the parameters it might need.

The syntax for the **mlx_loop_hook** () function is similar to the previous ones, but the given function will be called when no event occurs, and is not bound to a specific window.

When it catches an event, the MiniLibX calls the corresponding function with fixed parameters:

```
expose_hook(void *param);
key_hook(unsigned int keycode, void *param);
mouse_hook(unsigned int button, unsigned int x, unsigned int y, void *param);
loop_hook(void *param);
```

These function names are arbitrary. They here are used to distinguish parameters according to the event. These functions are NOT part of the MiniLibX.

param is the address specified in the mlx_*_hook calls. This address is never used nor modified by the MiniLibX. On key and mouse events, additional information is passed: *keycode* tells you which key is pressed (just try to find out :)), (*x* , *y*) are the coordinates of the mouse click in the window, and *button* tells you which mouse button was pressed.

Going further with events

The MiniLibX provides a much generic access to other available events. The *mlx.h* include define **mlx_hook()** in the same manner mlx_*_hook functions work. The event and mask values will be taken from the historical X11 include file "X.h". Some Wayland and MacOS events are mapped to these values when it makes sense, and the mask may not be used in some configurations.

See source code of the MiniLibX to find out how it will call your own function for a specific event.

Got any suggestions?

If you find any errors or have any new ideas for improving this repository, feel free to open an Issue or Pull Request, or contact me at my email address: nora@defitero.com