# Introduction

## Introduction

MiniLibX is a tiny graphics library which allows you to do the most basic things for rendering something in screens without any knowledge of X-Window and Cocoa. It provides so-called simple window creation, a questionable drawing tool, half-ass image functions and a weird event management system.

## About X-Window

X-Window is a network-oriented graphical system for unix. For example this is used when connecting to remote desktops. One of the most common examples of such implementation would be TeamViewer.

## About macOS

macOS handles the graphical access to its screen, however to access this, we must register our application to the underlying macOS graphical framework that handles the screen, windowing system, keyboard and mouse.

Getting started

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:

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

To link with the required internal macOS API:

```makefile
$(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!

## 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:

```c
#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 <kbd>CTRL</kbd> + <kbd>C</kbd> 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!":

```c
#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:

```c
#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:

```c
#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:

```c
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:

```c
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:

```c
#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.

colors

Colors are presented in an int format. It therefore requires some tricky things in order to obtain an int which can contain the ARGB values.

## The color integer standard

We shift bits to use the TRGB format. To define a color, we initialize it as follows: `0xTTRRGGBB`, where each character represents the following:

Letter	Description
`T`	transparency
`R`	red component
`G`	green component

`B` | blue component

RGB colors can be initialized as above, a few examples would be:

Color	TRGB representation
red	`0x00FF0000`
green	`0x0000FF00`
blue	`0x000000FF`

## Encoding and decoding colors

We can use two methods to encode and decode colors:

- bitshifting
- char/int conversion

### BitShifting

Since each byte contains `2^8 = 256` values ([1 byte = 8 bits](https://www.google.com/search?q=size+bytes+to+bit)),
and RGB values range from 0 to 255, we can perfectly fit a integer (as an
int is 4 bytes). In order to set the values programatically we use `bitshifting`.
Let's create a function which does precisely that for us, shall we?

```c
int create_trgb(int t, int r, int g, int b)
{
return (t << 24 | r << 16 | g << 8 | b);
}
```

Because ints are stored from right to left, we need to bitshift each value the according amount of bits backwards. We can also do the exact opposite and retrieve integer values from a encoded TRGB integer.

```c
int get_t(int trgb)
{
return ((trgb >> 24) & 0xFF);
}

int get_r(int trgb)
{
return ((trgb >> 16) & 0xFF);
}

int get_g(int trgb)
{
return ((trgb >> 8) & 0xFF);
}
```

int get_b(int trgb)

{

return (trgb & 0xFF);

}

~~~

Char/int conversion

Since each byte contains `2^8 = 256` values ([1 byte = 8 bits](https://www.google.com/search?q=size+bytes+to+bit)),

and RGB values range from 0 to 255, we can perfectly fit a `unsigned char` for each TRGB parameters `{T, R, G, B}`

(char is 1 byte) and fit a `int` for the TRGB value (int is 4 bytes). In order to set the values programatically

we use type converting.

```c

int create_trgb(unsigned char t, unsigned char r, unsigned char g, unsigned char b)

{

return (*(int *)(unsigned char [4]){b, g, r, t});

}

unsigned char get_t(int trgb)

{

return (((unsigned char *)&trgb)[3]);

}

unsigned char get_r(int trgb)
~~~

```
{

return (((unsigned char *)&trgb)[2]);

}

unsigned char get_g(int trgb)

{

return (((unsigned char *)&trgb)[1]);

}

unsigned char get_b(int trgb)

{

return (((unsigned char *)&trgb)[0]);

}
```

To understand the conversion you can refere to the table bellow, where `0x0FAE1` is the variable address of `int trgb`.

| Address | char | int |
| ------- | :-------------: | :-------------: |
| 0x0FAE1 | unsigned char b | int trgb |
| 0x0FAE2 | unsigned char g | [allocated] |
| 0x0FAE3 | unsigned char r | [allocated] |
| 0x0FAE4 | unsigned char t | [allocated] |

## Test your skills!

Now that you understand the basics of how the colors can be initialized, get comfy and try creating the following color manipulation functions:
- `add_shade` is a function that accepts a double (distance) and a int (color) as arguments, 0 will add no shading to the color, whilst 1 will make the color completely dark. 0.5 will dim it halfway, and .25 a quarter way. You get the point.
- `get_opposite` is a function that accepts a int (color) as argument and that will invert the color accordingly.

# Events

## Introduction

Events are the foundation of writing interactive applications in MiniLibX. It is therefore of essence that you fully comprehend this chapter as this will be of use in your future graphical projects.

All hooks in MiniLibX are nothing more than a function that gets called whenever a event is triggered. Mastering all these events won't be neccessary, however, we will quickly go over each X11 event accordingly.

### MacOS version

Note: On MacOS - Cocoa (AppKit) and OpenGL - version, minilibx has partial support of X11 events and doesn't support X11 mask (x_mask argument of mlx_hook is useless, keep it at 0).

Supported events:

```c
enum {
ON_KEYDOWN = 2,
ON_KEYUP = 3,
ON_MOUSEDOWN = 4,
ON_MOUSEUP = 5,
ON_MOUSEMOVE = 6,
ON_EXPOSE = 12,
ON_DESTROY = 17
};


// usage:
mlx_hook(vars.win, ON_DESTROY, 0, close, &vars);
```

## X11 interface

X11 is the library that is used alongside of MiniLibX. It therefore is no secret that this header is very useful for finding all the according events of MiniLibX.

### X11 events

There are a number of events to which you may describe.

Key	Event		Key	Event		Key	Event
`02`	KeyPress		`14`	NoExpose		`26`	CirculateNotify
`03`	KeyRelease		`15`	VisibilityNotify		`27`	CirculateRequest
`04`	ButtonPress		`16`	CreateNotify		`28`	PropertyNotify
`05`	ButtonRelease		`17`	DestroyNotify		`29`	SelectionClear
`06`	MotionNotify		`18`	UnmapNotify		`30`	SelectionRequest
`07`	EnterNotify		`19`	MapNotify		`31`	SelectionNotify
`08`	LeaveNotify		`20`	MapRequest		`32`	ColormapNotify
`09`	FocusIn		`21`	ReparentNotify		`33`	ClientMessage
`10`	FocusOut		`22`	ConfigureNotify		`34`	MappingNotify
`11`	KeymapNotify		`23`	ConfigureRequest		`35`	GenericEvent
`12`	Expose		`24`	GravityNotify		`36`	LASTEvent
`13`	GraphicsExpose		`25`	ResizeRequest			

If you can't figure out what some events are, don't worry, because you probably won't need them. If you do, go read [the documentation of each X11 events](https://tronche.com/gui/x/xlib/events/).

### X11 masks

Each X11 event, also has a according mask. This way you can register to only one key when it triggers, or to all keys if you leave your mask to the default. Key masks therefore allow you to whitelist / blacklist events from your event subscriptions. The following masks are allowed:

Mask	Description		Mask	Description
`0L`	NoEventMask		`(1L<<12)`	Button5MotionMask
`(1L<<0)`	KeyPressMask		`(1L<<13)`	ButtonMotionMask
`(1L<<1)`	KeyReleaseMask		`(1L<<14)`	KeymapStateMask
`(1L<<2)`	ButtonPressMask		`(1L<<15)`	ExposureMask
`(1L<<3)`	ButtonReleaseMask		`(1L<<16)`	VisibilityChangeMask
`(1L<<4)`	EnterWindowMask		`(1L<<17)`	StructureNotifyMask
`(1L<<5)`	LeaveWindowMask		`(1L<<18)`	ResizeRedirectMask
`(1L<<6)`	PointerMotionMask		`(1L<<19)`	SubstructureNotifyMask
`(1L<<7)`	PointerMotionHintMask		`(1L<<20)`	SubstructureRedirectMask
`(1L<<8)`	Button1MotionMask		`(1L<<21)`	FocusChangeMask
`(1L<<9)`	Button2MotionMask		`(1L<<22)`	PropertyChangeMask
`(1L<<10)`	Button3MotionMask		`(1L<<23)`	ColormapChangeMask
`(1L<<11)`	Button4MotionMask		`(1L<<24)`	OwnerGrabButtonMask

## Hooking into events

### mlx_hook

Hooking into events is one of the most powerful tools that MiniLibX provides. It allows you to register to any of the aforementioned events with the call of a simple hook registration function.

To achieve this, we call the function `mlx_hook`.

```c
```

void mlx_hook(mlx_win_list_t *win_ptr, int x_event, int x_mask, int (*f)(), void *param)

```

*Some version of mlbx doesn't implement `x_mask` and whatever the value there will be no mask.*

### Prototype of event functions

Event functions have a different prototype depending of the hooking event.

| Hooking event | code | Prototype |
| ------------- | :--: | ------------------- |
| ON_KEYDOWN | 2 | `int (*f)(int keycode, void *param)` |
| ON_KEYUP* | 3 | `int (*f)(int keycode, void *param)` |
| ON_MOUSEDOWN* | 4 | `int (*f)(int button, int x, int y, void *param)` |
| ON_MOUSEUP | 5 | `int (*f)(int button, int x, int y, void *param)` |
| ON_MOUSEMOVE | 6 | `int (*f)(int x, int y, void *param)` |
| ON_EXPOSE* | 12 | `int (*f)(void *param)` |
| ON_DESTROY | 17 | `int (*f)(void *param)` |

**Has mlx_hook alias.*

### Hooking alias

Minilibx api has some alias hooking function:
```

- `mlx_expose_hook` function is an alias of mlx_hook on expose event (`12`).
- `mlx_key_hook` function is an alias of mlx_hook on key up event (`3`).
- `mlx_mouse_hook` function is an alias of mlx_hook on mouse down event (`4`).

### Example

For example instead of calling `mlx_key_hook`, we can also register to the `KeyPress` and `KeyRelease` events. Lets take a look:

```c
#include <mlx.h>

typedef struct s_vars {
void *mlx;
void *win;
} t_vars;

int close(int keycode, t_vars *vars)
{
mlx_destroy_window(vars->mlx, vars->win);
return (0);
}

int main(void)
{
```

```
t_vars vars;

vars.mlx = mlx_init();
vars.win = mlx_new_window(vars.mlx, 1920, 1080, "Hello world!");
mlx_hook(vars.win, 2, 1L<<0, close, &vars);
mlx_loop(vars.mlx);
}
```

Here we register to the `KeyPress` event with the according `KeyPressMask`. Now whenever we press a key, the window will be closed.

## Test your skills!

Now that you have a faint idea of how all of this works, we encourage you to make the hook handlers. Whenever the:
- <kbd>ESC</kbd> key is pressed, your window should close.
- window is resized, you should print something in your terminal.
- the red cross is clicked, your window should close.
- you press a key longer than x seconds, you should print something in your terminal.
- mouse enters the window, you should print `Hello!` in your terminal, when it leaves, you should print `Bye!`.

Hooks

In computer programming, the term *hooking* covers a range of techniques used to alter or augment the behaviour of an operating system, of applications, or of other software components by intercepting function calls or messages or events passed between software components. Code that handles such intercepted function calls, events or messages is called a hook.

## Introduction

Hooking is used for many purposes, including debugging and extending functionality. Examples might include intercepting keyboard or mouse event messages before they reach an application, or intercepting operating system calls in order to monitor behavior or modify the function of an application or another component. It is also widely used in benchmarking programs, for example to measure frame rate in 3D games, where the output and input is done through hooking.

Simply put, it is therefore not weird that hooking is the backbone of MiniLibX.

## Hooking into key events

Hooking may sound difficult, but it really is not. Let's take a look shall we?

```c
#include <mlx.h>
#include <stdio.h>

typedef struct s_vars {
void *mlx;
void *win;

} t_vars;



int key_hook(int keycode, t_vars *vars)

{

printf("Hello from key_hook!\n");

return (0);

}



int main(void)

{

t_vars vars;



vars.mlx = mlx_init();

vars.win = mlx_new_window(vars.mlx, 640, 480, "Hello world!");

mlx_key_hook(vars.win, key_hook, &vars);

mlx_loop(vars.mlx);

}
```

We have now registered a function that will print a message whenever we press a key. As you can see, we register a hook function with `mlx_key_hook`. However in the background it simply calls the function `mlx_hook` with the appropriate X11 event types. We will discuss this in the next chapter.

## Hooking into mouse events

<img align="right" src="res/mouse-schema.png">

Also, you can hook mouse events.

```c
mlx_mouse_hook(vars.win, mouse_hook, &vars);
```
Mouse code for MacOS:

- Left click: 1
- Right click: 2
- Middle click: 3
- Scroll up: 4
- Scroll down : 5

## Test your skills!

Now that you have a faint idea of what hooks are, we will allow you to create a few of your own. Create hook handlers that whenever:

- a key is pressed, it will print the key code in the terminal.
- the mouse if moved, it will print the current position of that mouse in the terminal.
- a mouse is pressed, it will print the angle at which it moved over the window to the terminal.

Loops

## Introduction

Now that you finally understand the basics of the MiniLibX library, we will start with drawing a tiny animation in the window. For this we will be using two new functions, namely `mlx_loop` and `mlx_loop_hook`.

Loops are a feature of MiniLibX where it will continue to call your hook registered in `mlx_loop_hook` to render new frames, which you obviously have to pass to the window yourself.

## Hooking into loops

To initiate a loop, we call the `mlx_loop` function with the `mlx` instance as only parameter, take a look:

```c
#include <mlx.h>

int main(void)
{
void *mlx;

mlx = mlx_init();
mlx_loop(mlx);
}
```

This will do nothing of course as we have no loop hook registered, therefore we will not be able to write anything to our frame.

To do this, you will have to create a new window and use the mutations that we described in the [Getting Started](./getting_started.html) chapter. We assume that your knowledge of that is proficient and that you will be able to pass your parameters accordingly. An example whiteboarded version of it could look as follows:

```c
#include <mlx.h>

int render_next_frame(void *YourStruct);

int main(void)
{
void *mlx;

mlx = mlx_init();
mlx_loop_hook(mlx, render_next_frame, YourStruct);
mlx_loop(mlx);
}
```

```

Now for each frame it requires, it will call the function `render_next_frame` with the parameter `YourStruct`. This will persist through multiple calls if it is a pointer, so use that to your advantage!

## Test your skills!

Now that you understand how to register your own rendering function, we suggest that you create the following renderers:
- Render a moving rainbow that shifts through all colors (screen turns red, becomes green and then becomes blue, then starts all over again).
- Create a circle that you can move accross your screen using your WASD keys.

# Images

## Introduction

Images are a very important tool in MiniLibX in order to embrace its full potential. These functions will allow you to read files directly into a image object. This is very useful for textures or sprites of course.

## Reading images

To read from a file to a image object, you need either the XMP or PNG format. In order to read we can call the functions `mlx_xpm_file_to_image` and `mlx_png_file_to_image` accordingly. Do mind that `mlx_png_file_to_image`
```

currently leaks memory. Both functions accept exactly the same parameters and their usage is identical.

Now, lets read from an image, shall we?

```c
#include <mlx.h>

int main(void)
{
void *mlx;
void *img;
char *relative_path = "./test.xpm";
int img_width;
int img_height;

mlx = mlx_init();
img = mlx_xpm_file_to_image(mlx, relative_path, &img_width, &img_height);
}
```

If the `img` variable is equal to `NULL`, it means that the image reading has failed. It will also set the `img_width` and `img_height` accordingly, which is ideal when reading sprites.

## Test your skills!

Now that you know how to read from files, lets get started on making more graphical stuff ;). Create the following programs:

- Import a cursor of your choice and allow it to roam within the window freely.
- Import a texture and replicate it accross your entire window.

Sync

## What is sync?

As mentioned before, you could manage your own frame buffering with MLX, however this is fairly tedious and time consuming. Besides, we use more memory and our frames need to constantly be FULLY updated. This is not very efficient hence we need to avoid it at all costs.

From the 2020 MLX version, you will be able to synchronize your own frames, which should make the hacky multiple images for screen buffering no longer required.

## Using sync

We have three defines that we need to understand first:

```c
#define MLX_SYNC_IMAGE_WRITABLE 1
#define MLX_SYNC_WIN_FLUSH_CMD 2
#define MLX_SYNC_WIN_CMD_COMPLETED 3
```

```
int mlx_sync(int cmd, void *ptr);
```

`mlx_sync` ought to be called with the defined command codes. The first one, `MLX_SYNC_IMAGE_WRITABLE` will buffer all subsequential calls to an image (`ptr` is a pointer to the MLX image object). If you want to propagate changes, you will have to flush the window in which the image is present, using `MLX_SYNC_WIN_FLUSH_CMD` and the window you want to flush as a `ptr`.

## Test your skills!

Now that you have a faint idea of what `mlx_sync` can do, grab your previous tiny circle-game that you made in [loops](./loops.html) and add `mlx_sync` to your rendering!

Epilogue

Now that you have finally completed the entire tutorial, you can finally say that you are a true master on MiniLibX. We have also created documentation for each function in case you quickly need to look it up or forgot what the according prototype was [over here](./prototypes.html).

If you think something is missing from this tutorial, feel free to hit us up with a pull request and we will check it ASAP.

Good luck with your projects!

Prototype

# Introduction

MiniLibX is a tiny graphics library which allows you to do the most basic things for rendering something. This can vary from making a copy of Wolfenstein, to presenting complicated data in a simple form.

It is truly recommended to catch up on bitwise operands if you have no clue what they are.

## Initialization functions

These are the standard functions that are almost always required to even start using MiniLibX.

### mlx_init

Initializes the MLX library. Must be called before ANY other function. Will return `NULL` if initialization failed.

```c
```

/*
** Initialize mlx.
**
** @return void* the mlx instance
*/
void *mlx_init();

### mlx_new_window

Creates a new window instance. It will return a window instance pointer.
This should be saved for future reference.

```c
/*
** Create a new window.
**
** @param void *mlx_ptr the mlx instance pointer;
** @param int size_x the width of the window;
** @param int size_y the height of the window;
** @param char *title the title of the window;
** @return void* the window instance pointer.
*/
void *mlx_new_window(void *mlx_ptr, int size_x, int size_y, char *title);
```

### mlx_clear_window

Clears the current window. This is not a recommended function to use. Instead it is recommended to use the function `mlx_put_image_to_window` with a recycled image that you have cleared.

```c
/*
** Clear the provided window.
**
** @param void *mlx_ptr the mlx instance pointer;
** @param void *win_ptr the window instance pointer;
** @return int has no return value (bc).
*/
int mlx_clear_window(void *mlx_ptr, void *win_ptr);
```

### mlx_destroy_window

Destroys a window instance accordingly.

```c
/*
** Destroy a window instance.
**
** @param void *mlx_ptr the mlx instance;
** @param void *win_ptr the window instance;
** @return int has no return value (bc).
```

```
*/
int mlx_destroy_window(void *mlx_ptr, void *win_ptr);
```

## Util functions

These are functions that can help you with conversions and pixel writing.

### mlx_get_color_value

Get the color value accordingly from a int. This is useful for converting a self-declared int before writing it to certain bits.

```c
/*
** Get the color value.
**
** @param void *mlx_ptr the mlx instance;
** @param int color the int color (0xTTRRGGBB);
** @return uint the converted color.
*/
uint mlx_get_color_value(void *mlx_ptr, int color);
```

### mlx_pixel_put

Puts a pixel on the screen. This function is NOT recommended for use. It will lock the window output, force a refresh and a recalculation. It is therefore suggested to render a image and push that using the `mlx_put_image_to_window` function. You can find more about that in the [Getting Started](./getting_started.html) chapter.

```c
/*
** Put a pixel on the screen.
**
** @param void *mlx_ptr the mlx instance pointer;
** @param void *win_ptr the window instance pointer;
** @param int x the x coordinate of the pixel to draw;
** @param int y the y coordinate of the pixel to draw;
** @param int color the color of the pixel to draw (0xTTRRGGBB);
** @return int has no return value (bc).
*/
int mlx_pixel_put(void *mlx_ptr, void *win_ptr, int x, int y, int color);
```

### mlx_string_put

Puts a string on the location (x,y) in the given window.

```c
/*
** Put a string in the window.
**
```

** @param void *mlx_ptr the mlx instance;

** @param int x the x location;

** @param int y the y location;

** @param int color the font color;

** @param char *string the text to write;

** @return int has no return value (bc).

*/

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

~~~

Image functions

Image object related functions. These will provide effective methods to mutate frames one-by-one.

mlx_new_image

Creates a new MLX compatible image. This is the recommended way to buffer the image you are rendering. It will accept a pointer to your MLX instance and requires a width and height. Will return a reference pointer to the image.

```c

/*

** Create a new MLX compatible image.

**

** @param void *mlx_ptr the mlx instance pointer;
~~~

```c
** @param int width the width of the image to be created;
** @param int height the height of the image to be created;
** @return void* the image instance reference.
*/
void *mlx_new_image(void *mlx_ptr,int width,int height);
```

### mlx_get_data_addr

Gets the memory address of the given image. Memory of images is weird. It will set the line size in your given pointer. To get or set the value of the pixel (5, 100) in an image size of (500, 500), we would need to locate the position as follows:

```c
int pos = (y * size_line + x * (bits_per_pixel / 8));
```

Here we multiply size_line by `y` as we need to skip `y` lines (and yes, line size is not equal to the amount of pixels in a line). We then add the remaining `x` units multiplied by `bits_per_pixel / 8` to align with the final location.

To modify each pixel with the correct color, we need to do some more fancy stuff. As we need to align the bits before writing, we need to do the following for the best result:

```c
char *mlx_data_addr = mlx_get_data_addr();
*(unsigned int *)mlx_data_addr = color;
```

The function prototype is as follows:

```c
/*
** Gets the data address of the current image.
**
** @param void *img_ptr the image instance;
** @param int *bits_per_pixel a pointer to where the bpp is written;
** @param int *size_line a pointer to where the line is written;
** @param int *endian a pointer to where the endian is written;
** @return char* the memory address of the image.
*/
char *mlx_get_data_addr(void *img_ptr, int *bits_per_pixel, int *size_line, int
*endian);
```

### mlx_put_image_to_window

Puts an image to the given window instance at location (x,y). This is the recommended way to write large amounts of graphical data in one go. Do mind that when changing the memory of the locations, it will be displayed directly on the window.

```c
/*
** Put an image to the given window.
**
** @param void *mlx_ptr the mlx instance;
** @param void *win_ptr the window instance;
** @param int x the x location of where the image ought to be placed;
** @param int y the y location of where the image ought to be placed;
** @return int has no return value (bc).
*/
int mlx_put_image_to_window(void *mlx_ptr, void *win_ptr, void *img_ptr, int x, int
y);
```

### mlx_destroy_image

Destroys an image instance accordingly.

```c
/*
** Destroy an image instance.
**
** @param void *mlx_ptr the mlx instance;
** @param void *img_ptr the image instance;
** @return int has no return value (bc).
*/
```

int mlx_destroy_image(void *mlx_ptr, void *img_ptr);

## Hooks

These functions will allow you to hook into the MiniLibX functions. This is core functionality and is required to use the library effectively. Please look at the [Hooks](./hooks.html) chapter if you have no clue what this means.

### mlx_mouse_hook

Hook into mouse events. This will trigger every time you click somewhere in the given screen. Do mind that currently these mouse events barely work, it is therefore suggested to not use them.

```c
/*
** Hook into mouse events.
**
** @param void *win_ptr the window instance;
** @param int (*f)() the handler function, will be called as follows:
** (*f)(int button, int x, int y, void *param);
** @param void *param the parameter to give on each event;
** @return int has no return value (bc).
*/
int mlx_mouse_hook(void *win_ptr, int (*f)(), void *param);
```

### mlx_key_hook

Hook into key events. This will trigger every time a key is pressed in a focused window. Unfocused windows will not register any key events.

```c
/*
** Hook into key events.
**
** @param void *win_ptr the window instance;
** @param int (*f)() the handler function, will be called as follows:
** (*f)(int key_code, void *param);
** @param void *param the parameter to give on each event;
** @return int has no return value (bc).
*/
int mlx_key_hook(void *win_ptr, int (*f)(), void *param);
```

### mlx_expose_hook

Has no defined behaviour.

### mlx_loop_hook

Hook into the loop.

```c
/*
** Hook into each loop.
**
** @param void *mlx_ptr the mlx instance;
** @param int (*f)() the handler function, will be called as follows:
** (*f)(void *param);
** @param void *param the parameter to give on each event;
** @return int has no return value (bc).
*/
int mlx_loop_hook(void *mlx_ptr, int (*f)(), void *param);
```

### mlx_loop

Loop over the given MLX pointer. Each hook that was registered prior to this will be called accordingly by order of registration.

```c
/*
** Loop over the given mlx pointer.
**
** @param void *mlx_ptr the mlx instance;
** @return int has no return value (bc).
*/
int mlx_loop(void *mlx_ptr);
```

## Image conversions

These are functions that are useful for loading sprites or even saving images.

### mlx_xpm_to_image

Converts xpm data to a new image instance.

```c
/*
** Converts xpm data to a new image instance.
**
** @param void *mlx_ptr the mlx instance;
** @param char **xpm_data the xpm data in a 2 dimensional char array;
** @param int *width a pointer to where the width ought to be written;
** @param int *height a pointer to where the height ought to be written;
** @return void* the image instance, and NULL in case of error.
*/
void *mlx_xpm_to_image(void *mlx_ptr, char **xpm_data, int *width, int *height);
```

### mlx_xpm_file_to_image

Converts an xpm file to a new image instance.

```c
/*
** Convert an xpm file to a new image instance.
**
** @param void *mlx_ptr the mlx instance;
** @param char *filename the file to load;
** @param int *width a pointer to where the width ought to be written;
** @param int *height a pointer to where the height ought to be written;
** @return void* the image instance, and NULL in case of error.
*/
void *mlx_xpm_file_to_image(void *mlx_ptr, char *filename, int *width, int *height);
```

### mlx_png_file_to_image

Converts a png file to a new image instance.

```c
/*
** Convert a png file to a new image instance.
**
** @param void *mlx_ptr the mlx instance;
** @param char *filename the file to load;
** @param int *width a pointer to where the width ought to be written;
** @param int *height a pointer to where the height ought to be written;
** @warn mem_leak this function has a memory leak, try using xpm
** instead;
```

~~~~markdown
** @return void* the image instance.
*/
void *mlx_png_file_to_image(void *mlx_ptr, char *filename, int *width, int *height);
```

## Mouse functions

These functions will allow you to hide, show, move or get the mouse position.

### mlx_mouse_hide

Hides the mouse.

```c
/*
** Hide the mouse.
**
** @return int has no return value (bc).
*/
int mlx_mouse_hide();
```

### mlx_mouse_show

Shows the mouse.
~~~~

```c
/*
** Show the mouse.
**
** @return int has no return value (bc).
*/
int mlx_mouse_show();
```

### mlx_mouse_move

Moves the cursor to the given location.

```c
/*
** Move the cursor to the given location.
**
** @param void *win_ptr the window instance;
** @param int x the x location to move to;
** @param int y the y location to move to;
** @return int has no return value (bc).
*/
int mlx_mouse_move(void *win_ptr, int x, int y);
```

### mlx_mouse_get_pos

Gets the current mouse position on the window.

```c
/*
** Get the current mouse position on the window.
**
** @param void *win_ptr the window instance;
** @param int *x the pointer to write the x location to;
** @param int *y the pointer to write the y location to;
** @return int has no return value (bc).
*/
int mlx_mouse_get_pos(void *win_ptr, int *x, int *y);
```

## Key auto repeat

These functions will allow you to either enable or disable key autorepeat.

### mlx_do_key_autorepeatoff

Disable key auto repeat.

```c
/*
** Disable key auto repeat.
```

```c
**
** @param void *mlx_ptr the mlx instance;
** @return int has no return value (bc).
*/
int mlx_do_key_autorepeatoff(void *mlx_ptr);
```

### mlx_do_key_autorepeaton

Enable key auto repeat.

```c
/*
** Enable key auto repeat.
**
** @param void *mlx_ptr the mlx instance;
** @return int has no return value (bc).
*/
int mlx_do_key_autorepeaton(void *mlx_ptr);
```

## Un-categorized

### mlx_do_sync

```c
```

/*

** Synchronize frames of all windows in MLX.

**

** @param void *mlx_ptr the mlx instance;

** @return int has no return value (bc).

*/

int mlx_do_sync(void *mlx_ptr);

### mlx_get_screen_size

```c

/*

** Get the current screen size (because macOS is sheit)

**

** @param void *mlx_ptr the mlx instance;

** @param int *sizex the screen width;

** @param int *sizey the screen height

** @return int has no return value (bc).

*/

int mlx_get_screen_size(void *mlx_ptr, int *sizex, int *sizey);

```