Introduction to OpenMLX Basics

I- Introduction

OpenMLX : An Advanced C Extension of the Minilibx Graphics Library

During our studies at school 42, we had to use the Minilibx for various graphic projects. Although basic, this library has some limitations. For example, it uses the CPU instead of the GPU, which results in no-optimal performance and limits the possibilities for complex creation.

Why the name OpenMLX ?

The OpenMLX name was chosen because this extension is based on Minilibx and several functions are inspired or come from OpenGL, another graphics library.

Major Additions:

- Improved images management.
- Advanced window management.
- Optimized display management.

Is this allowed at school 42 ?

For now yes. All basic functions used to design this project are permitted by the all Graphics Projects Rules and meet school standards. You can build in some of these features, but there will be a lot to adapt because OpenMLX was designed to be used in its entirety.

OpenMLX aims to fill the gaps of Minilibx by providing better performance and more features, while remaining compliant with school 42 requirements.

II- Warning

OpenMLX is still under development (Work in Progress). Bugs may still occur and will be fixed as soon as possible. If you find minor (e.g. images not saving) or major (e.g. segfaults) bugs, you can report them. Just make sure the problem isn't with your code. (Sincerely)

For some school 42 graphics projects, like the **So_Long** project, you will need to add **"#define SO_LONG_PROJECT 1**". This disables rendering via mlx_pixel_put and instead uses a recast of mlx_pixel_put, which simulates its effect. Instead of inserting a pixel directly into the window, the function takes a 1x1 image, assigns it the correct color, and then inserts the image into the window. This is a circumvention of the rule prohibiting pixel-by-pixel rendering. Don't overdo it; This method is primarily intended for rendering player or collectibles.

OpenMLX is 4242% free and open source. It will never be paid or private.

III- Basic Functions

```
# define PI 3.14159265359
# define GRAPHICS_SYNC_DELAY 0

typedef struct ml_s
{
    void        *ptr;
    void        *win;
    int         width;
    int         height;
    int         (*(set_win_size))();
    int         (*(make_window))();
    void        (*(purge_window))();
    void        (*(new_purge_color))();
    void        (*(quit_window))();
    t_texture   texture;
    __uint32_t  purge_color;
}   t_ml;
```

GRAPHICS_SYNC_DELAY : Delay between each image insertion for the reconstructed function "**mlx_put_pixel**".

ptr : Pointer returned by « **mlx_init** ».

win : Pointer returned by « **mlx_new_window** ».

width : Width of the window.

height : Height of the window.

purge_color : Default color to clear window.

texture : Structure containing shaders (t_shaders) (list of registered images) and a shader counter (unsigned int shaders_count).

set_win_size(width, height) : Call the function « **_ml set_window_size** » to set the size of the window (return 1 if the process is correctly executed or 0 if he fail).

make_window(char title) : Call the function « **_ml_create_window** » to create the window with the defined size and initialize the different images by default.

purge_window : Call the function « **_ml_purge_window** » to clear the window with "**mlx_clear_window**" and display a default color.

new_purge_color(unsigned int color) : Call the function « **_ml_set_purge_color** » to change the default color when purging the window.

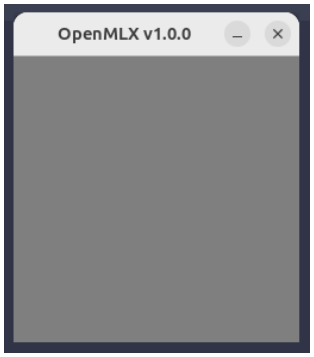
quit_window : Call the function "**_ml_quit**", deleting all images stored in the program, also calls "**mlx_destroy_window**" and "**mlx_destroy_display**", and frees the memory allocated to gmlx.

```
enum e_gmlx
{
    ACT_INIT    = 0,
    ACT_GET,
    ACT_FREE,
    ACT_MAX
};

t_ml    *gmlx(int e_gmlxact)
{
    static t_ml *lx;

    if (e_gmlxact == ACT_INIT && xalloc((void **)&lx, 1, sizeof(t_ml)))
    {
        lx->make_window = _ml_create_window;
        lx->set_win_size = _ml_set_window_size;
        lx->purge_window = _ml_purge_window;
        lx->quit_window = _ml_quit;
        lx->new_purge_color = _ml_set_purge_color;
        return (lx);
    }
    else if (e_gmlxact == ACT_GET && lx)
        return (lx);
    else if (e_gmlxact == ACT_FREE && lx)
        return (xfree((void **)&lx), NULL);
    return (NULL);
}
```

int xalloc(void buffer, size_t size, size_t nb) : Performs a calloc and checks that the memory allocation is successful. If it is, the static lx is allocated when gmlx(ACT_INIT) is called.

void xfree(void buffer) : Free the memory allocation and set the buffer pointer to NULL.

```
//Setup Exemple
int main(void)
{
    t_ml        *lx;

    lx = gmlx(ACT_INIT);
    if (lx)
    {
        lx->purge_color = 0x7F7F7F;
        if (lx->set_win_size(240, 240) && lx->make_window("OpenMLX v1.0.0"))
        {
            sleep(999);
            lx->quit_window();
        }
    }
}
```



IV- Images management

Load a local image:

```
//Setup Exemple

static void load_images(void)
{
    register_img("./rupies.xpm");
    register_img("./ground.xpm");
    register_img("./rock.xpm");
}

int main(void)
{
    t_ml        *lx;

    lx = gmlx(ACT_INIT);
    if (lx)
    {
        lx->purge_color = 0x7F7F7F;
        if (lx->set_win_size(240, 240) && lx->make_window("OpenMLX v1.0.0"))
        {
            load_images();
            sleep(999);
            lx->quit_window();
        }
    }
}
```

register_img(char path) : This function splits the path to obtain the image name. For example, ./texture/rupies.xpm becomes /rupies.xpm. If /rupies.xpm is already present in the list of shaders, the image registration (**register_img**) is not performed. Otherwise, it stores the different information of the image in the t_shaders structure.

```
typedef struct s_img
{
    void    *ptr;
    char    *addr;
    int     bpp;
    int     len;
    int     endian;
    int     width;
    int     height;
}   t_img;

typedef struct shader_s
{
    __uint32_t          file;
    int                 created;
    int                 is_stored;
    t_img               img;
    struct shader_s     *next;
}   t_shaders;

typedef struct texture_s
{
    t_shaders   shaders;
    __uint32_t  shaders_count;
}   t_texture;
```

Create an image inside the program:

```
typedef struct ui_s
{
    int         x;
    int         y;
    int         w;
    int         h;
    __uint32_t  color;
}   t_ui;

void    build_images(void)
{
    create_img((t_ui){0, 0, 40, 40, 0xFF0000}, fill_img_color, "red_square");
    create_img((t_ui){0, 0, 40, 40, 0xFF0000}, NULL, "empty");
}

/*

if (func == fill_img_color)
    func(&box.img, ui.color);
else if (func)
    func(&box.img, ui);

*/
```

fill_img_color(t_img *dest, __uint32_t color) filled the image '**dest'** by the color '**color'**.

If a function is given as a parameter, then once the initial creation of the image is completed, the '**func**' function is called with the generated image and the various parameters contained in the '**t_ui**' structure.

E.g: void build_logo(t_img *dest, t_ui ui);

Get a image :

```
void    get_img_usage(void)
{
    t_shaders   sh;

    create_img((t_ui){0, 0, 40, 40, 0xFF0000}, NULL, "empty");
    register_img("./rupies.xpm");
    sh = get_img("/rupies.xpm");
    if (sh)
    {
        ...
    }
    sh = get_img("empty");
    if (sh)
    {
        ...
    }
}
```

Print a image :

```
void    print_img_usage(void)
{
    t_shaders   sh;

    create_img((t_ui){0, 0, 40, 40, 0xFF0000}, NULL, "empty");
    register_img("./rupies.xpm");
    print_img((t_vec2){24, 23}, "/rupies.xpm");
    print_img((t_vec2){50, 78}, "empty");
}
```

print_img(t_vec2 xy, char *name) : Displays the image ‘**name**’ at the ‘**xy**’ origin (position).

Blend colors :

```
void    blend_colors_usage(void)
{
    __uint32_t  color;

    color = blend_colors(0xFFFFFF, 0xFF0000, 0.8);
    create_img((t_ui){0, 0, 40, 40, color}, fill_img_color, "red_square");
}
//__uint32_t    blend_colors(__uint32_t bkg, __uint32_t frg, float alpha)
```

blend_colors has various uses. For example, it can create a gradient between two colors by adjusting the intensity. The closer the alpha value is to 0.0f, the closer the color will be to the background color (**bkg**). Conversely, if the alpha value is close to 1.0f, the color will be closer to the foreground color (**frg**). However, its true utility is revealed in the **overlay_images** function, which manages transparency between two images.

Create an overlay between 2 images :

```
void    build_overlay(void)
{
    create_img((t_ui){0, 0, 60, 60, 0x00}, NULL, "empty");
    register_img("./ground.xpm");//60x60
    register_img("./rock.xpm");//60x60
    overlay_images(&get_img("empty")->img, &get_img("/ground.xpm")->img, &get_img("/rock.xpm")->img, 0.6f);
    print_img((t_vec2){40, 40}, "empty");
}
//void  overlay_images(t_img *d, t_img *b, t_img *f, float a)
```

To use the **overlay_images** function, you will need three different images. The first one, '**d**', is the image that will be used as the destination. The second one, '**b**', is the background. The third one, '**f**', is the foreground. Finally, '**alpha**' will be the transparency of the third image.

V- Conclusion

Here concludes the introduction to the various functions present in OpenMLX. Overall, these functions pertain to image and window management. Other functions will be added, such as hook management, etc.

The functions not presented here belong to more advanced basics. To use them properly, it is necessary to have knowledge of vertex/vertices, rendering, texture mapping, etc., or at least a minimal understanding of OpenGL.