Introduction aux Bases d'OpenMLX

I- Introduction

OpenMLX : Une Extension Avancée en C de la Bibliothèque Graphique Minilibx

Durant notre cursus à l'école 42, nous avons été amenés à utiliser la Minilibx pour divers projets graphiques. Bien que basique, cette bibliothèque présente certaines limitations. Par exemple, elle utilise le CPU au lieu du GPU, ce qui entraîne des performances non optimales et limite les possibilités de création complexe.

Pourquoi le nom OpenMLX ?

Le nom OpenMLX a été choisi car cette extension est basée sur la Minilibx et plusieurs fonctions sont inspirées ou proviennent d'OpenGL, une autre bibliothèque graphique.

Ajouts Majeurs :

- Gestion améliorée des images.
- Gestion avancée des fenêtres (Window).
- Gestion optimisée de l'affichage.

Est-ce autorisé dans le cursus 42 ?

Pour le moment, oui. Toutes les fonctions de base utilisées pour concevoir ce projet sont autorisées par les règles des projets graphiques de 42 et respectent les normes de l'école. Vous pouvez intégrer certaines de ces fonctions, mais il y aura beaucoup d'éléments à adapter, car OpenMLX a été conçue pour être utilisée dans son intégralité.

OpenMLX vise à combler les lacunes de la Minilibx en offrant une meilleure performance et plus de fonctionnalités, tout en restant conforme aux exigences académiques de 42.

II- Avertissements

OpenMLX est toujours en cours de développement (Work in Progress). Des bugs peuvent encore survenir et seront corrigés dès que possible. Si vous trouvez des bugs mineurs (par exemple, des images qui ne s'enregistrent pas) ou majeurs (par exemple, des segfaults), vous pouvez les signaler. Assurez-vous simplement que le problème ne provient pas de votre code. (Cordialement)

Pour certains projets graphiques de 42, comme le projet **So_Long**, vous devrez ajouter **« #define SO_LONG_PROJECT 1** ». Cela désactive le rendu via mlx_pixel_put et utilise à la place une refonte de mlx_pixel_put, qui simule son effet. Au lieu d'insérer un pixel directement dans la fenêtre, la fonction utilise une image de 1x1, lui attribue la bonne couleur, puis insère l'image dans la fenêtre. C'est littéralement un contournement de la règle interdisant le rendu pixel par pixel. N'en abusez pas ; cette méthode est principalement destinée au rendu du joueur ou des objets collectables.

OpenMLX est a 4242% gratuite et open source. Elle ne sera jamais payante ni privée.

III- Fonctions de base

```
# 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 : Délai entre chaque insertion d'images pour la fonction reconstruite « **mlx_put_pixel** ».

ptr : Pointeur retourné par « **mlx_init** ».

win : Pointeur retourné par « **mlx_new_window** ».

width : Largeur de la fenêtre.

height : Hauteur de la fenêtre.

purge_color : Couleur par défaut pour effacer la fenêtre.

texture : Structure contenant des shaders (**t_shaders**) (liste chaînée des images enregistrées) et un compteur de shaders (unsigned int shaders_count).

set_win_size(width, height) : Appelle « **_ml_set_window_size** » pour définir la taille de la fenêtre (retourne 1 si l'opération est effectuée avec succès, 0 sinon).

make_window(char title) : Appelle « **_ml_create_window** » pour créer la fenêtre avec la taille définie et initialiser les différentes images par défaut.

purge_window : Appelle « **_ml_purge_window** » pour effacer la fenêtre avec «**mlx_clear_window** » et afficher une couleur par défaut.

new_purge_color(unsigned int color) : Appelle « **_ml_set_purge_color** » pour redéfinir la couleur par défaut lors de la purge de la fenêtre.

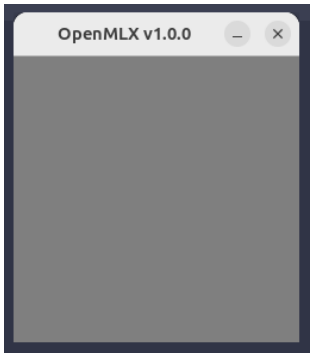
quit_window : Appelle « **_ml_quit** », supprimant toutes les images stockées dans le programme, et appelle également « **mlx_destroy_window** » et « **mlx_destroy_display** », et libère la mémoire allouée à gmlx.

```
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) : Effectue un calloc et vérifie que l'allocation mémoire est correctement effectuée. Si c'est le cas, le static lx est alloué si on appelle gmlx(ACT_INIT).

void xfree(void buffer) : Libère l'allocation mémoire et met le pointeur buffer à 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- Gestion des images

Charger une image local :

```
//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) : Cette fonction divise le chemin pour obtenir le nom de l'image. Par exemple, ./texture/rupies.xpm devient /rupies.xpm. Si /rupies.xpm est déjà présent dans la liste des shaders, l'enregistrement de l'image (**register_img**) ne s'effectue pas. Sinon, elle stocke les différentes informations de l'image dans la structure t_shaders

```
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;
```

Créer une image a l'intérieur du programme :

```
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) : remplie l'image '**dest'** par la couleur '**color'**.

Si une fonction est donnée en paramètre, alors une fois la création initiale de l'image terminée, la fonction '**func**' est appelée avec l'image générée et les différents paramètres contenus dans la structure '**t_ui**'.

Ex : void build_logo(t_img *dest, t_ui ui) ;

Récupérer une 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)
    {
        ...
    }
}
```

Afficher une 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) : Affiche l'image '**name**' a la position correspondante a '**xy**'.

Mélanger des couleurs :

```
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 a différents usages. Par exemple, on peut créer un dégradé entre deux couleurs en ajustant l'intensité. Plus la valeur de l'alpha est proche de 0.0f, plus la couleur sera proche de la couleur de fond (bkg). Inversement, si l'alpha est proche de 1.0f, la couleur sera plus proche de la couleur de premier plan(frg). Cependant, son véritable usage se révèle dans la fonction **overlay_images**, qui permet de gérer la transparence entre deux images.

Créer un overlay entre 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)
```

Pour utiliser la fonction **overlay_images** on auras besoin de 3 images différentes la première '**d**' est l'image qui sera utiliser comme destination, la deuxième '**b**' est le background (l'arrière plan), la troisième '**f**' est le foreground (premier plan) et enfin l'alpha sera la transparence de la troisième image.

V- Conclusion

Voila qui conclus l'introduction aux différentes fonctions présentes dans OpenMLX. Globalement, ces fonctions concernent la gestion des images et des fenêtres. D'autres fonctions seront ajoutées, telles que la gestion des hooks, etc.

Les fonctions non présentées ici appartiennent à des bases plus avancées. Pour les utiliser correctement, il est nécessaire d'avoir des notions sur les vertex/vertices, le rendering, le texture mapping, etc., ou du moins une connaissance minimale d'OpenGL.