

Start Date :

End Date :

# CUB3D

Ce projet est inspiré du jeu éponyme mondialement connu, considéré comme le premier FPS jamais développé. Il vous permettra d'explorer la technique du ray-casting. Votre objectif est de faire une vue dynamique au sein d'un labyrinthe, dans lequel vous devrez trouver votre chemin.

---

## Contexte et Histoire :

### Wolfenstein 3D:

- Développé par **ID Software** et les ultra-célèbres **John Carmack et John Romero**;
- Publié en **1992** par **Apogee Software**;
- Le premier “First Person Shooter” dans l'histoire du jeu-vidéo;
- L'ancêtre des jeux tels que **Doom** (Id Software, 1993), **Doom II** (Id Software, 1994), **Duke Nukem 3D** (3D Realm, 1996) et **Quake** (Id Software, 1996) qui sont des pierres angulaires additionnelles dans le monde du jeu vidéo.

## Objectifs

Cub3D est une aire de jeu remarquable pour explorer les applications pratiques des mathématiques sans avoir à en comprendre les spécificités. Avec l'aide des nombreux documents disponibles sur internet, vous utiliserez les mathématiques en tant qu'outil de création d'algorithmes élégants et efficaces.

| Comme tous les autres projets, les principaux objectifs sont   |                  |                              |                          |
|--|------------------|------------------------------|--------------------------|
| Rigueur  | Utilisation du C | Utilisation d'algos basiques | Recherche d'informations |
| Comme ce projet est un projet de <b>design graphiques</b> , Cub3D nous permettra de travailler nos talents de designer |                  |                              |                          |
| Fenêtres   | Couleurs         | Événements                   | Formes                   |

---

## Partie Obligatoire

|                               |  |
|-------------------------------|--|
| Norme                         | Le projet doit être à <b>la norme</b> , y compris les fichiers bonus.  |
| Allocation de Mémoire         | Toute mémoire allouée sur la heap doit être libérée lorsque nécessaire.<br>Aucun leak également.   |
| Compilation                   | <b>-Wall -Wextra -Werror.</b>  |
| Makefile                      | <b>\$(NAME), all, clean, fclean et re</b> (si les bonus sont réalisés, le Makefile doit également contenir une règle bonus).   |
| Libft                         | <b>Autorisée</b>   |
| Fonctions Externes Autorisées | <b>Open, Close, Read, Write, Printf, Malloc, free, Perror, Streeror, Exit.</b><br>Toutes les fonctions de la <b>lib math</b> (-lm man man 3 math).<br>Toutes les fonctions de la <b>MinilibX</b> . |

**Description:** Le but est de créer une **représentation graphique 3D "réaliste"** que nous pourrions avoir au sein d'un **labyrinthe** en utilisant une **vue subjective**. Cette représentation doit être créée en utilisant les principes du **ray-casting**.

### Contraintes:

- La MinilibX doit être utilisé, soit dans la version disponible sur votre OS, ou depuis ses sources (si elle est utilisée avec les sources, les mêmes règles que la libft s'appliquent);
- La gestion des fenêtres doit être parfaite : gestion de la minimisation, du passage d'une autre fenêtre, etc;
- L'affichage des différentes textures selon si les murs sont face nord, sud, est ou ouest est obligatoire (mais le choix de texture est libre).

## Conseils, Règles à Suivre, Précision

1. Le programme doit être capable d'afficher des objets (sprite) au lieu d'un mur.
2. Le programme doit être capable d'avoir des couleurs différentes pour le sol et le plafond.
3. En prévision du jour ou **deepthought** aura des yeux pour évaluer le projet, le programme doit pouvoir sauver la première image rendue au format bmp lorsque le second argument est “**--save**”.
4. Si il n'y a pas de second argument, le programme affiche l'image dans une fenêtre tout en respectant les règles suivantes:
  - Les touches flèches du gauche et droite du clavier doivent permettre de faire une rotation de la caméra (regarder à gauche et à droite);
  - Les touches W, A, S et D doivent permettre de déplacer la caméra (déplacement du personnage);
  - Appuyer sur la touche ESC doit fermer la fenêtre et quitter le programme proprement;
  - Cliquez sur la croix rouge de la fenêtre doit fermer la fenêtre et quitter le programme proprement.
  - Si la taille de fenêtre demandée dans la map est plus grande que celle de l'écran, la taille de fenêtre doit être celle de l'écran.
  - L'utilisation d'images de la MinilibX est fortement recommandée.
5. Le programme doit prendre en argument un fichier de description de scène avec pour extension .cub :
  - La map doit être composée d'uniquement ces 4 caractères :

|              |   |   |
|--------------|---|---|
| 0            | Pour les espaces vides  | Cette simple map doit être valide :<br><br>111111<br>100101<br>102001<br>1100N1<br>111111 |
| 1            | Pour les murs   |   |
| 2            | Pour un objet   |   |
| N, S, E ou W | Qui représentent la position de départ du joueur et son orientation |   |
  - La map doit être fermée/entourée de murs, sinon le programme doit renvoyer une erreur;
  - Mise à part la description de la map, chaque type d'élément peut être séparée par une ou plusieurs lignes vides;
  - La description de la carte sera toujours en dernier dans le fichier, le reste des éléments peut être dans n'importe quel ordre;
  - Sauf la map elle-même, les informations de chaque élément peuvent être séparées par un ou plusieurs espace(s);
  - La carte doit être partagée en accord avec ce qui est présent dans le fichier (les espaces sont une partie valable de la carte, c'est au programmeur de gérer correctement) : le programmeur doit pouvoir parser n'importe quelle sorte de carte, tant qu'elle respecte les règles de carte.
6. Pour chaque élément, le premier caractère est l'identifiant (un ou deux caractères), suivi de toutes les informations spécifiques à l'élément dans un ordre strict tel que :

| Résolution         | Identifiant | Taille de rendu axe x         | Taille de rendu axe y |
|--------------------|-------------|-------------------------------|-----------------------|
|                    | R           | 1920                          | 1080                  |
| North Texture      | Identifiant | Chemin vers la texture nord   |                       |
|                    | NO          | ./path_to_the_north_texture   |                       |
| South Texture      | Identifiant | Chemin vers la texture sud    |                       |
|                    | SO          | ./path_to_the_south_texture   |                       |
| West Texture       | Identifiant | Chemin vers la texture ouest  |                       |
|                    | WE          | ./path_to_the_west_texture    |                       |
| East Texture       | Identifiant | Chemin vers la texture est    |                       |
|                    | EA          | ./path_to_the_sprite_texture  |                       |
| Sprite Texture     | Identifiant | Chemin vers la texture sprite |                       |
| (objet)            | S           | ./path_to_the_sprite_texture  |                       |
| Couleur du Sol     | Identifiant | Couleur R, G, B range [0,255] |                       |
|                    | F           | 220,100,0                     |                       |
| Couleur de plafond | Identifiant | Couleur R, G, B range [0,255] |                       |
|                    | C           | 0,255,255                     |                       |

### **Exemple minimaliste de scène de la partie obligatoire .cub :**

\*Si un problème de configuration de n'importe quelle type est rencontré dans le fichier, le programme doit quitter et renvoyer "Error\n" suivi d'un message d'erreur explicite de votre choix.

```
R 1920 1080
NO ./path_to_the_north_texture
SO ./path_to_the_south_texture
WE ./path_to_the_west_texture
EA ./path_to_the_east_texture

S ./path_to_the_sprite_texture
F 220,100,0
C 225,30,0

    1111111111111111111111111111
    1000000000011000000000000001
    1011000001110000002000001
    10010000000000000000000000000001
111111111011000001110000000000001
1000000000110000011101111111111
111101111111101110000010001
1111011111111011101010010001
11000000110101011100000010001
100020000000000001100000010001
100000000000000001101010010001
11000000111010101111011110N0111
11110111 1110101 101111010001
11111111 1111111 111111111111
```

---

### **Recherches faites sur les sites suivant :**

[https://www.youtube.com/watch?v=js7HW65MmNw&list=PL0H9-oZI\\_QOHM34HvD3DiGmwmj5X7GvTW&ab\\_channel=BlogCr%C3%A9ationdeJeuxVid%C3%A9o](https://www.youtube.com/watch?v=js7HW65MmNw&list=PL0H9-oZI_QOHM34HvD3DiGmwmj5X7GvTW&ab_channel=BlogCr%C3%A9ationdeJeuxVid%C3%A9o)

<https://github.com/iciamyplant/Cub3d-Linux>

[https://www.youtube.com/watch?v=gID\\_FKfncZI](https://www.youtube.com/watch?v=gID_FKfncZI)

## I. Etape 1 : Le Parsing (de la map) ----->> Structure

### Rappel du malloc d'un tableau de chaînes de caractères

```
char **liste;
char *ptr
liste = malloc(sizeof(char*) * nbrdechaines)
liste[i] = malloc(sizeof(char) * ft_strlen(str))
```

Pour débuter le projet il faut se lancer dans le parsing de la map. Pour assurer un bon parsing, il faut :

- **Faire un tableau de chaines de caracteres a partir de la map;**
- Pour le malloc de ce tableau il faut **déterminer la taille de la chaîne la plus longue ainsi que les caractères qui la composent** pour ensuite **malloquer le tableau a partir du nombre de char\*** qu'il y a dans la map et **répéter** cette opération autant de fois qu'il y a de chaînes de caractères;
- **Parser a partir d'un tableau de char/int a double entrée;**
- Checker que la map soit **entourée de murs**;
- Vérifier que la map est bien **composée des caractères 0, 1 et 2**;
- Vérifier que le joueur se trouve bien dans la map (s'il y a N, S, E ou W) et s'il est présent, **remplacer son emplacement par 0 tout retenant la position et la direction du joueur dans une variable**;
- **Remplacer tous les espaces par des murs ;**
- **Ajouter des murs au bout pour que la taille de la chaîne soit suffisamment grande.**

Toutefois, il faut faire attention à certains détails pour assurer le bon fonctionnement du parsing :

- Est-ce qu'il **manque quelque chose** ? R, NO, SO, S... ?
- Est-ce qu'il y a **deux fois le même élément** ? Deux R, deux NO... ?
- Est-ce que la **résolution en int est plus grande que int max** ?
- Est-ce que la **résolution est à virgule** ? Est-ce qu'il possède un autre caractère ?
- Est-ce que la **résolution est à 3 chiffres, un seul chiffre ou égale à 0** ?
- Est-ce que **F (sol) ou C (plafond) manquent ou à un chiffre en trop** ?
- Est-ce que F ou C a une **virgule en trop ou en moins** ?
- Est-ce que F ou C est un **int supérieur à int max** ? (Si oui, il faut renvoyer une erreur)..
- Est-ce que F ou C a un **chiffre supérieur à 255** ?
- Est-ce qu'il y a un **mauvais identifiant** ? Un X ou lieu d'un R, un E au lieu de EA ?
- Il y a-t-il une **ligne vide** dans la map (attention, à ne pas confondre, les informations de chaque élément peuvent être séparées par un ou plusieurs espaces ainsi les espaces sont des partie valable de la carte) ?
- Il y a-t-il un **caractère incorrect** dans la map ? Un 4, par exemple ?
- **La map est-elle ouverte ?**
- **La map est-elle avant un autre élément ?**
- **Est-ce qu'il y a une map tout court ?**
- Est-ce qu'il n'y a pas de joueur ? Ou y a-t-il **plusieurs joueurs** ?

## **II. Etape 2 : La MinilibX**

---

### **1. Introduction :**

MinilibX est une **petite bibliothèque** qui permet de faire les choses les plus élémentaires pour rendre quelque chose sur des écrans sans aucune connaissance de X-Window et Cocoa. Il fournit ce qu'il appelle la **création de fenêtres simples**, un outil de dessin douteux, des fonctions d'image semi-assises et un système de gestion d'événement étrange.

---

### **2. Compilation sur MacOs :**

Parce que MinilibX **nécessite Appkit et X11**, nous devons les liens en conséquence. Pour éviter un processus de compilation compliqué, il est conseillé d'ajouter la règle suivante au **Makefile** (cela suppose d'avoir déjà la source mlx dans un répertoire nommé mlx à la racine du projet) :

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

Afin de créer un lien avec les APO MacOs internes requises, il faut ajouter la ligne suivant au makefile :

```
$(NAME): $(OBJ)  
$(CC) -Lmlx -lmlx -framework OpenGL -framework Appkit -o $(NAME)
```

**Attention :** La libmlx.dylib doit être dans le même répertoire que la cible de construction car il s'agit d'une bibliothèque dynamique.

---

### **3. Initialisation :**

Avant de pouvoir faire quoique ce soit avec la MinilibX, il faut include **<mlx.h>** header afin de pouvoir accéder aux fonctions et la **fonctions mlx\_init doit être exécutée** afin d'établir **la connexion avec le bon système graphique** (la fonction retournera a **void \*** qui détient l'emplacement de notre instance **MLX** actuelle).

Afin d'**initialiser MinilibX** on pourra :

```
#include <mlx.h>

int main(void)
{
    void *mlx;
    mlx = mlx_init();
}
```

A partir de là, on peut **initialiser une petite fenêtre** qui restera ouverte tant que tu ne la ferme pas en utilisant CTRL+C dans le terminal. Pour cela, il faut **appeler la fonction mlx\_new\_window**, qui retourne un pointeur sur la fenêtre qui vient d'être créée. **Il est possible de donner à cette fenêtre un hauteur, une largeur et un titre.** Pour lancer le rendu de la fenêtre, il faudra appeler la fonction **mlx\_loop**.

#### Creer une fenetre avec une largeur de 1080, une hauteur de 1920 et que se nomme "Hello World!"

```
#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);
}
```

#### 4. Ecrire des pixels sur une image :

A présent, il est possible de **push des pixels sur la fenêtre créée**, a note que l'obtention de ces pixels dépend entièrement de nous bien qu'il a des façons plus optimiser de les obtenir.

Il est possible d'**utiliser la fonction mlx\_pixel\_put mais cette solution reste très lente.** En effet, cette fonction tente de push instantanément les pixels sur la fenêtre sans attendre que le cadre soit entièrement rendu. C'est pour cela, qu'il faudra **bufferiser tous les pixels sur une image qui seront entre pousser vers celle-ci** :

Dans un premier temps, il est important de comprendre **quel type d'image requiert mlx**. Lors de l'initiation d'une image, il sera nécessaire de faire passer quelques pointeurs sur lesquels sera écrit quelques variables importantes. **La première démarche à suivre est celle du bpp (bits per pixel** puisque les pixels sont des int, qui sont habituellement de 4 bits, toutefois cela peut différer si nous gérons un petit endian). Ainsi, pour **initialiser une image** :

```
#include <mlx.h>

int main(void)
{
    void *mlx;
    void *img;

    mlx = mlx_init();
    img = mlx_new_image(mlx, 1920, 1080);
}
```

Maintenant qu'on possède une image, il s'agit d'écrire **des pixels dessus/par-dessus**. Pour cela, il faut **obtenir l'adresse mémoire sur laquelle nous allons muter les octets en conséquence**. L'obtention de cette adresse va comme suit :

```
#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);
}
```

Maintenant que nous possédons l'adresse mémoire de l'image et avant de passer au pixels il faut bien comprendre que les octets ne sont pas alignés, de fait cela signifie que line\_length diffère de la largeur réelle de la fenêtre. Ainsi, **il faut toujours calculer le décalage mémoire en utilisant la longueur de ligne définie par mlx\_get\_data\_addr** :

```
int offset = (y * line_length + x * (bits_per_pixel / 8));
```

A présent que nous savons où l'on peut écrire, il devient simple d'écrire **une fonction qui se comportera de la même façon que mlx\_pixel\_put mais qui sera bien plus rapide** :

```
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;
}
```

**Attention :** Cela causera des problèmes puisque une image est représentée en temps réel dans une fenêtre, ainsi changer la même image engendre plusieurs déchirements de fenêtre lors de l'écriture. Ainsi, **il est conseillé de créer une ou plusieurs images pour tenir vos cours temporairement**. De fait, **il est possible d'écrire une image temporaire pour éviter de devoir écrire sur l'image actuelle**.

---

## 5. Push une image sur une fenêtre :

Maintenant que l'on sait créer une image, il faut la push sur la fenêtre afin de la voir. Pour illustrer cela nous allons voir **comment écrire un pixel rouge à (5,5) et l'afficher sur la fenêtre** :

```

#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);
}

```

\*0x00F0000 est la représentation hexadécimale de ARGB(0,255,0,0);

---

## **6. Les événements (quand on clique sur une touche) :**

La MinilibX dispose d'une fonction nommée "**mlx\_hook**" permettant d'ajouter une fonction de gestion d'événement à son code (un int) associé, dont le protocole est le suivant :

|  |   |
|--|---|
| int mlx_hook(void *win_ptr, int x_event, int x_mask, int (*funct)(), void *param); |   |
| win_ptr  | New window  |
| x_event  | Le "masque" de l'événement que l'on veut gérer (lire manuel X)  |
| param  | Un paramètre divers que vous pouvez passer à la fonction qui gère l'événement   |
| funct  | La fonction qu'on lance quand l'événement se passe (il y a différents types de fonctions selon si c'est un mouvement de la souris, un keypress, etc.) |

La MinilibX supporte certaines fonctions pour quelques types d'événement précis dont les prototypes sont :

|                      |           |
|----------------------|-----------|
| Evenement de type... | Prototype |
|----------------------|-----------|

|                                       |   |
|---------------------------------------|---|
| KeyPress                              | int funct(int keycode, void *param);              |
| KeyRelease                            | int funct(int keycode, void *param);              |
| ButtonPress (souris)                  | int funct(int button, int x, int y, void *param); |
| MotionNotify<br>(mouvement de souris) | int funct(int x, int y, void *params);            |
| Tous les autres...                    | int funct(void *params);                          |

---

## 7. La fonction “mlx\_loop\_hook” et “mlx\_put\_image\_to\_window”:

**La syntaxe de la fonction mlx\_hoop\_hook() est identique à celle de mlx\_hoop() mais celle-ci se lance en continue.** Il faut penser à mettre mlx\_put\_image\_to\_window dans la fonction qui se trouve dans loop\_hook, sinon l'image ne s'imprime pas :

```
int mlx_loop_hook ( void *mlx_ptr, int (*funct_ptr)(), void *param );
int mlx_put_image_to_window ( void *mlx_ptr, void *win_ptr, void *img_ptr, int x, int y );
```

### **III. Etape 3 : La Minimap**

---

#### **1. En quoi consiste la Minimap ?**

L'étape du la Minimap **consiste à utiliser le parsing** de l'étape précédente (le tableau char\*\*) pour **créer une Minimap avec les 0, les 1 et les 2 chacun d'une couleur**.

Il faudra faire des pixels (10 pixels) pour qu'on voit correctement la Minimap, pouvoir faire bouger le personnage avec les flèches dans la Minimap et checker si la case sur laquelle je vais me déplacer est un mur ou pas (si == '0' : si oui, je peux me déplacer dessus, sinon non).

---

#### **2. Comment s'y prendre ?**

Il faudra utiliser les fonctions de la MinilibX : comme **mlx\_init**, **mlx\_new\_window** et **mlx\_loop**. Plus précisément, il faudra :

| Fonctions   | Utilisation   |
|---|---|
| <b>mlx_hook</b>   | <b>Tourne en fond pour les key_press et les key_release</b> qui permet de récupérer si une touche est appuyée ou non. |
| <b>mlx_get_data_addr</b>                                  | Récupérer l'adresse de l'image et écrire des pixels dedans.   |
| <b>mlx_loop_hook + la fonction qui imprime la Minimap</b> | Pour que dès qu'il y a une keypress la minimap s'adapte.  |
| <b>mlx_put_image_to_window</b>                            | Pour put l'image une fois adapter.  |

#### IV. Etape 4 : Les Keys

---

##### 1. Les keys que doit contenir le projet :

| Action                              | Conséquences                     | Keycodes on Mac            |
|-------------------------------------|----------------------------------|----------------------------|
| Appuyer sur <b>escape</b>           | Quitte proprement (avec exit(0)) |                            |
| Appuyer sur <b>flèche de gauche</b> | Rotation gauche                  | define ROTATE_LEFT<br>123  |
| Appuyer sur <b>flèche de droite</b> | Rotation droite                  | define ROTATE_RIGHT<br>124 |
| Appuyer sur <b>W</b>                | Avancer                          | define FORWARD_W_Z<br>13   |
| Appuyer sur <b>S</b>                | Reculer                          | define BACK_S_S 1          |
| Appuyer sur <b>A</b>                | Déplacer à gauche                | define RIGHT_D_D 2         |
| Appuyer sur <b>D</b>                | Déplacer à droite                | define LEFT_A_Q 0          |

---

## **2. Keys and MiniLibX**

### **a. Introduction :**

**Les événements sont à la base de l'écriture d'applications interactives dans MiniLibX.** Tous les **hooks** dans la MiniLibX ne sont rien de plus qu'une **fonction qui est appelée à chaque fois qu'un événement est déclenché.**

### **b. X11 events :**

**X11 est la bibliothèque utilisée avec MiniLibX,** à partir de cela l'on peut décrire un certains nombres d'événements :

|    |                |    |                  |    |                  |
|----|----------------|----|------------------|----|------------------|
| 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 | SelectionNotify  |
| 07 | EnterNotify    | 19 | MapNotify        | 31 | SelectionRequest |
| 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    |    |                  |

### c. X11 Masks :

Chaque événement X11 a également un **masque correspondant**. De cette, on peut **enregistrer sur une seule touche lorsqu'elle se déclenche**, ou sur toutes les touches si on laisse le masque par défaut.

Les masques de clés **permettent donc de mettre en liste blanche / noire les événements des abonnements aux événements**. Les masques suivants sont autorisés :

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

#### d. Se connecter à un événement :

La connexion à des événements est l'un des outils les plus puissants fournis par la MiniLibX. Il vous permet de vous inscrire à l'un des événements susmentionnés avec l'appel d'une simple fonction d'enregistrement de crochet.

Par exemple, au lieu d'appeler mlx\_key\_hook, on peut également s'inscrire aux événements KeyPress et KeyRelease. Pour ce faire, on appelle la fonction mlx\_hook :

```
#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);
}

int main(void)
{
    t_vars vars;

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

Ici, on s'inscrit à l'événement KeyPress avec le KeyPressMask correspondant. Désormais, chaque fois que l'on appuie sur une touche, la fenêtre se ferme.

```
int mlx_hook(void *win_ptr, int x_event, int x_mask, int (*funct)(), void
*param);
```

## V. Etape 5 : Le Raycasting I (Walls)

---

### 1. Introduction :

Le **Raycasting** est une **technique de rendu permettant de créer une perspective 3D dans une carte 2D**. A l'époque où les ordinateurs étaient plus lents, il n'était pas possible d'exécuter de vrais moteurs 3D en temps réel : le raycasting était la première solution. Le **Raycasting** peut aller très vite, car **un seul calcul doit être fait pour chaque ligne verticale de l'écran**.

Le jeu le plus connu qui utilise cette technique est bien sûr **Wolfenstein 3D**. Le moteur de diffusion de rayons de Wolfenstein 3D était très limité, ce qui lui permettait de fonctionner sur un même ordinateur 286 : **tous les murs ont la même hauteur et sont des carrés orthogonaux sur une grille 2D**.



Editeur de carte Wold3D

Des choses comme les escaliers, les sauts ou les différences de hauteur sont impossibles à faire avec ce moteur. Des jeux ultérieurs tels que Doom et Duke Nukem 3D utilisaient également le raycasting, mais des moteurs beaucoup plus avancés étaient utilisés et permettaient des murs en pente, différentes hauteurs, sols et plafonds texturés, murs transparents, etc... Les **sprites** (ennemis, objets et goodies), eux, sont des **images 2D**, mais les sprites ne sont pas abordés dans ce didacticiel pour le moment.

**Le Raycasting n'est pas la même chose que le Raytracing !**

Le Raycasting est une technique de rendu semi-3D rapide qui fonctionne en temps réel même sur des calculatrices graphiques à 4 MHz, tandis que le raytracing est une technique de rendu réaliste qui prend en charge les reflets et les ombres dans des vraies scènes 3D, et ce n'est que récemment que les ordinateurs sont devenus assez rapides pour le faire en temps réel pour un niveau raisonnablement élevé qui permet une résolutions et scènes complexes.

---

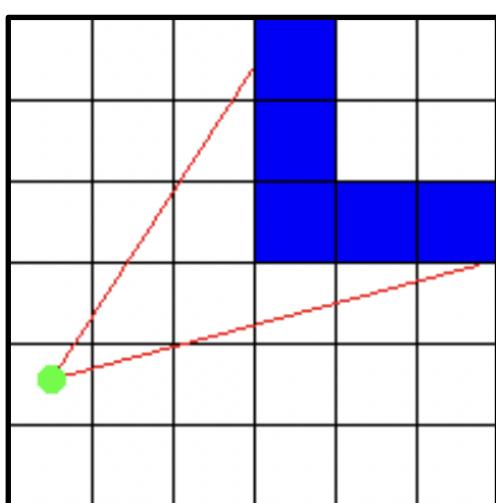
## 2. The Basic Idea :

L'idée de base du raycasting est la suivante :

**La carte est une grille carré 2D, et chaque carré peut être soit 0 (= pas de mur), soit une valeur positive (= un mur avec une certaine couleur ou texture).**

**Pour chaque x de l'écran** (c'est-à-dire pour chaque bande verticale de l'écran), il faut envoyer un rayon qui commence à l'emplacement du joueur et avec une direction qui dépend à la fois de la direction de recherche du joueur et de la coordonnée x de l'écran.

Ensuite, il faut laisser ce rayon avancer sur la carte 2D, jusqu'à ce qu'il atteigne un carré de carte qui est un mur. S'il heurte un mur, alors il faut calculer la distance de ce point de vie au joueur et utiliser cette distance pour calculer à quelle hauteur ce mur doit être dessiné sur l'écran et plus il est proche, plus il semble élevé : Ce sont tous des calculs 2D.



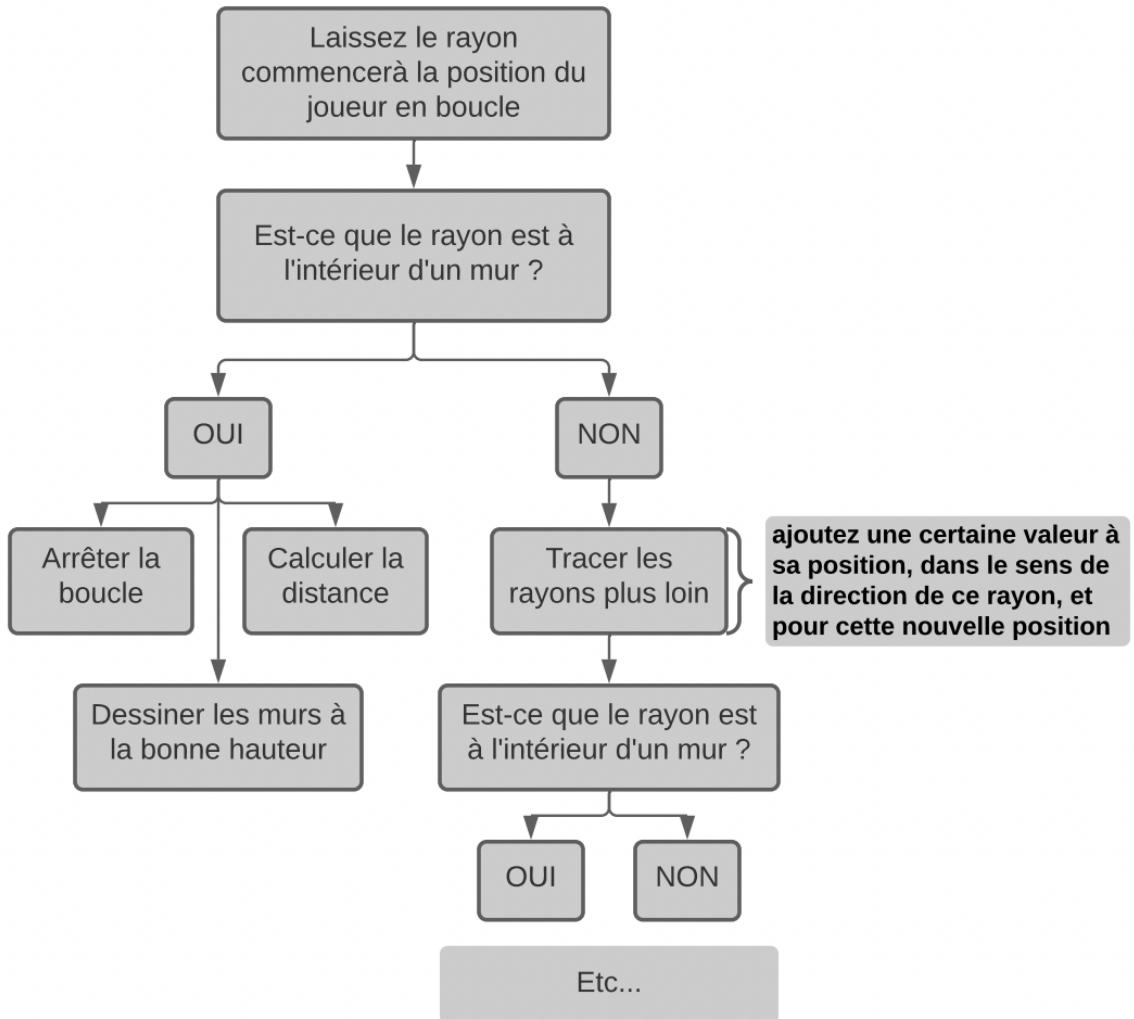
Cette image montre un aperçu de haut en bas de deux de ces rayons (en rouge) qui commencent au joueur (le point vert) et frappent les murs bleus.

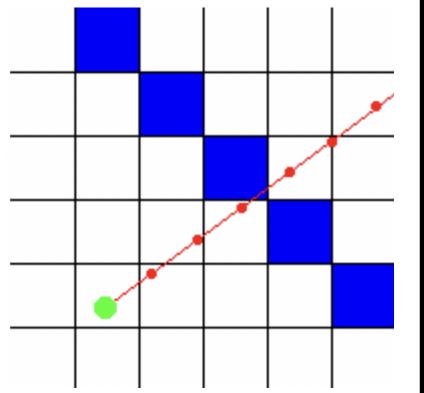
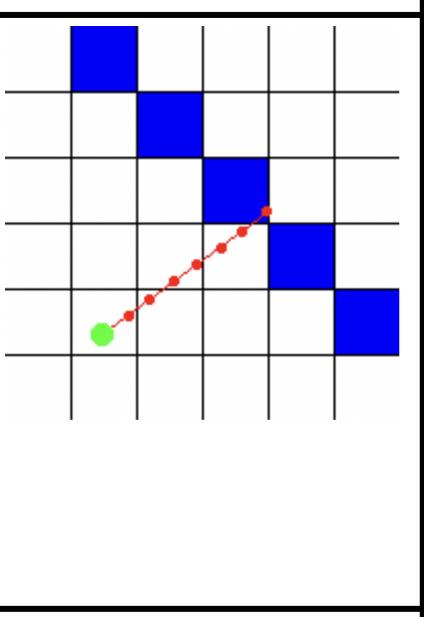
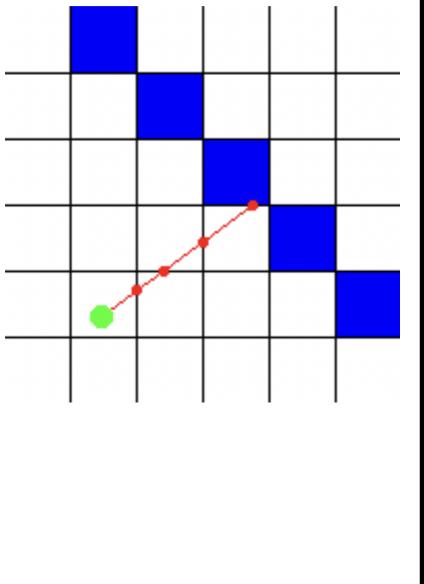
Pour trouver le premier mur qu'un rayon rencontre sur son chemin, il faut le laisser commencer à la position du joueur, puis tout le temps, et vérifier si le rayon est ou non à l'intérieur d'un mur (coup). Dès lors, la boucle peut s'arrêter et il faut à présent calculer la distance et dessiner les murs à la bonne hauteur.

Si la position du rayon n'est pas dans un mur, il faut la tracer plus loin : c'est-à-dire ajoutez une certaine valeur à sa position, dans le sens de la

direction de ce rayon, et pour cette nouvelle position, pour ensuite vérifier à nouveau si elle est à l'intérieur d'un mur ou non. Il faut continuer à faire cela jusqu'à ce qu'un mur soit finalement touché.

Un humain peut voir immédiatement où le rayon frappe le mur, mais il est impossible de trouver quel carré le rayon frappe immédiatement avec une seule formule, car un ordinateur ne peut vérifier qu'un fini de positions sur le rayon. De nombreux lanceurs de rayons ajoutent une valeur constante au rayon à chaque étape, mais il y a alors une chance qu'il rate un mur!



|   |   |
|---|---|
|    | <p>Par exemple, ici, avec ce rayon rouge, sa position a été vérifiée à chaque point rouge.</p> <p>Comme on peut le voir, le rayon passe directement à travers le mur bleu, mais l'ordinateur ne l'a pas détecté, car il n'a vérifié que les positions avec les points rouges. <b>Plus on vérifie de positions, plus il y a de calculs, moins il y a de chances que l'ordinateur ne détecte pas de mur.</b></p>  |
|   | <p>Ici, la distance de pas a été réduite de moitié, alors maintenant il détecte que le rayon a traversé un mur, bien que la position ne soit pas tout à fait correcte.</p> <p>Pour une précision infinie avec cette méthode, une taille de pas infinitiment petite, et donc un nombre infini de calculs seraient nécessaires !</p> <p>Heureusement, il existe une autre méthode qui ne nécessite que très peu de calculs et qui détecte pourtant chaque mur : <b>l'idée est de vérifier chaque côté d'un mur que le rayon rencontrera. Il faut donner à chaque carré une largeur de 1, donc chaque côté d'un mur est une valeur entière et les endroits intermédiaires ont une valeur après le point.</b></p> |
|  | <p>Maintenant, la taille du pas n'est pas constante, cela dépend de la distance au côté suivant.</p> <p>Comme on peut le constater, ici, le <b>rayon frappe le mur exactement où nous le voulons</b>. Pour cela, on utilise un <b>algorithme basé sur DDA</b>.</p> <p><b>“Digital Differential Analysis”</b>, est un algorithme rapide généralement utilisé sur les grilles carrées pour trouver les carrés d'une ligne de frappe, comme pour dessiner une ligne sur un écran, qui est une grille de pixels carrés).</p> <p>Ainsi, cette technique peut être utilisée pour trouver les carrés de la carte que le rayon atteint, et arrêter l'algorithme une fois qu'un carré qui est un mur est touché.</p>   |

Certains traceurs de rayons fonctionnent avec des angles euclidiens pour représenter la direction du joueur et des rayons, et ainsi déterminer le champ de vision avec un autre angle.

Il est cependant beaucoup plus facile de travailler avec des vecteurs et une caméra à la place :

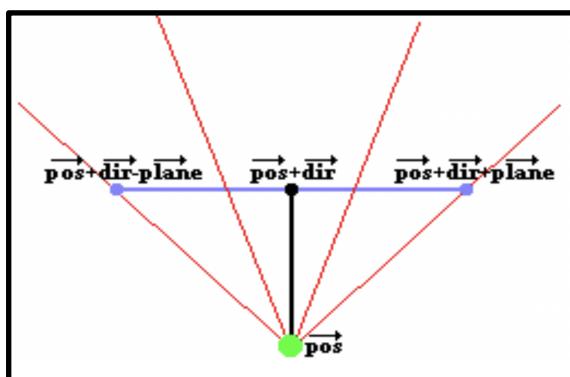
- La position du joueur est toujours un vecteur (une coordonnée x et y);
- La direction devient elle aussi ;
- De fait, la direction est maintenant déterminée par deux valeurs (les coordonnées x et y de la direction).

Un vecteur de direction doit être vu comme suit : si l'on trace un ligne dans la direction que le joueur regarde, à travers la position du joueur, alors chaque point de la ligne est la somme de la position du joueur et un multiple de la direction vecteur. La longueur d'un vecteur de direction n'a pas vraiment d'importance, seulement sa direct. La multiplication de x et y par la même valeur change la longueur mais garde la même direction.

Cette méthode avec des vecteurs nécessite également un **vecteur supplémentaire**: le **vecteur du plan de la caméra**. Dans un vrai moteur 3D, il y aussi un plan de caméra, ce plan est vraiment en plan 3D donc deux vecteurs (**u** et **v**) sont nécessaires pour le représenter. Le lancer de rayons se produit cependant dans une carte 2D, donc ici, le plan de la caméra n'est pas vraiment un plan, mais une ligne qui est représentée avec un seul vecteur.

**Le plan de la caméra doit toujours être perpendiculaire au vecteur de direction.** Le plan de la caméra représente la surface de l'écran de l'ordinateur, tandis que le vecteur de direction y est perpendiculaire et pointe à l'intérieur de l'écran.

**La position du joueur, qui est un point unique, est un point devant le plan de la caméra.** Un certain rayon d'une certaine coordonnée X de l'écran est alors le rayon qui commence à cette position du joueur, et passe par cette position sur l'écran ou donc le plan de la caméra.



Cette image  
une telle

représente justement  
caméra 2D

Le point vert est la position (vecteur "pos").

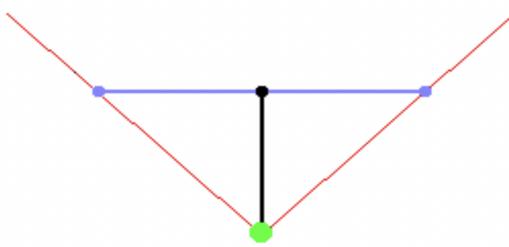
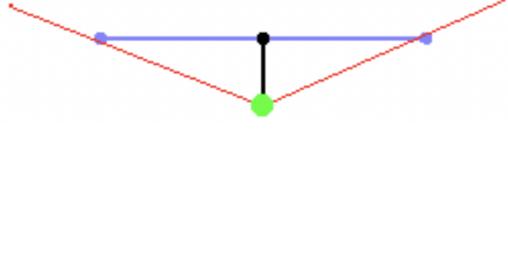
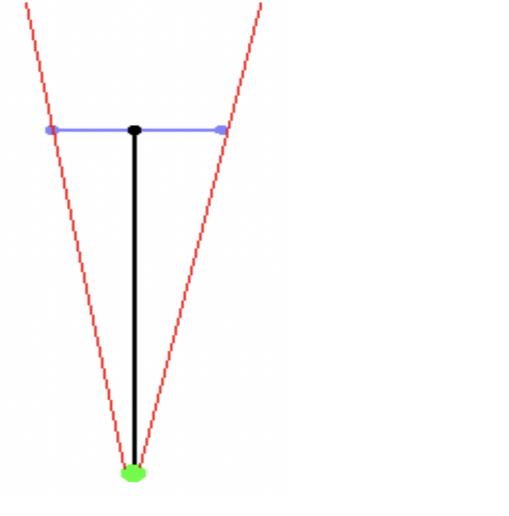
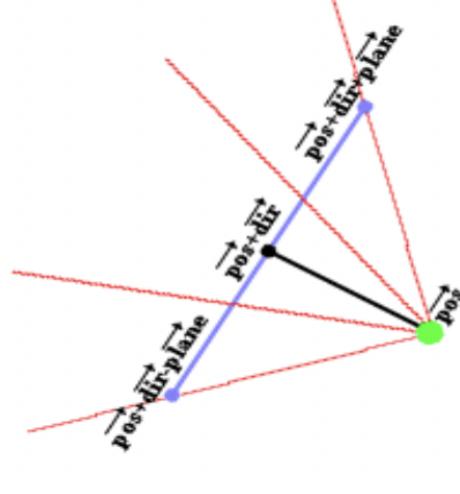
La ligne noire, se terminant par le point noir, représente le vecteur de direction (vecteur "dir"). De fait, la position du point noir est pos + dir.

La ligne bleu représente la plan complet de la caméra, le vecteur du point noir au point bleu droit représente le vecteur "plan", donc la position du point bleu droit est pos + dit + plan et la position du point bleu gauche est pos + dir - plan (ce sont tous des ajouts de vecteurs).

**Les lignes rouges de l'image sont quelques rayons. La direction de ces rayons se calcule facilement hors de la caméra : c'est la somme du vecteur direction de la caméra, et d'une partie du vecteur plan de la caméra.** Par exemple, le troisième rayon rouge sur l'image, passe par la partie droite du plan de la caméra au point environ  $\frac{1}{3}$  de sa longueur. La direction de ce rayon est donc  $\text{dir} + \text{plan} * \frac{1}{3}$  de sa longueur. Cette direction de rayon est le vecteur `rayDir`, et les composantes X et Y de ce vecteur sont ensuite utilisées par l'algorithme DDA.

**Les deux lignes extérieures sont la bordure gauche et droite de l'écran, et l'angle entre ces deux lignes est appelé le champ de vision ou FOV.**

**Le FOV est déterminé par le rapport de la longueur du vecteur de direction et de la longueur du plan.** Voici quelques exemples de différents FOV :

|   |   |
|---|---|
|    |   |
| <p>Si le vecteur direction et le vecteur plan caméra ont la même longueur, le FOV sera de <math>90^\circ</math></p>   | <p>Si le vecteur de direction est plus court que le plan de la caméra, le FOV sera supérieur à <math>90^\circ</math> (<math>180^\circ</math> est le maximum, si le vecteur de direction est proche de 0). On aura alors une <b>vision beaucoup plus large</b>, comme le zoom arrière.</p>   |
|   |    |
| <p>Si le vecteur de direction est beaucoup plus long que le plan de caméra, le FOV sera beaucoup plus petit que <math>90^\circ</math> et on aura une <b>vision très étroite</b>. Cependant, on verra tout plus détaillé et il y aura <b>moins de profondeur</b>, c'est donc la même chose que zoomer.</p> | <p>Lorsque le joueur tourne, la caméra doit aussi tourner, donc le vecteur de direction et le vecteur de plan doivent être également tournés. Ensuite, les rayons tourneront tous automatiquement.</p> <p>Pour faire pivoter un vecteur, il faut le multiplier avec la matrice de rotation: <math>\begin{bmatrix} \cos(a) &amp; -\sin(a) \\ \sin(a) &amp; \cos(a) \end{bmatrix}</math>.</p> |

### **3. Untextured Raycasting (exemple) :**

a. La carte du monde :

La carte du monde est un tableau 2D, où chaque valeur représente un carré. Si la valeur est 0, ce carré représente un carré vide pouvant être parcouru à pied, et si la valeur est supérieur à 0, il représente un mur avec une certaine couleur ou texture.

La carte déclarée ici est très petite, seulement 24 carrés sur 24 carrés, et est définie directement dans le code. Pour un vrai jeu, comme Wolfenstein 3D, on peut utiliser une carte plus grande et la charger à partir d'un fichier à la place. Tous les zéros de la grille sont des espaces vides, donc fondamentalement, on peut voir une très grande pièce, avec un mur autour d'elle (les valeurs 1), une petite pièce à l'intérieur (les valeurs 2), quelques piliers (les valeurs 3), et un couloir avec une pièce (les valeurs 4). A noter que ce code n'est pas encore à l'intérieur d'une fonction, il faut le placer avant le démarrage de la fonction principale.

---

## b. Déclaration des variables :

```
int main(int /*argc*/, char **/*argv*/[])
{
    double posX = 22, posY = 12; //x and y start position
    double dirX = -1, dirY = 0; //initial direction vector
    double planex = 0, planeY = 0.66; //the 2d raycaster version of camera plane

    double time = 0; //time of current frame
    double oldTime = 0; //time of previous frame
```

Quelques premières variables sont déclarées :

- **posX et posY représentent le vecteur de position du joueur;**
- **dirX et dirY représentent la direction du joueur;**
- **planX et planY représentent le plan de la caméra du joueur** (il faut bien s'assurer que le plan de la caméra est perpendiculaire à la direction, mais la longueur peut être modifiée);
- **Les variables time et oldTime seront utilisées pour stocker l'heure de l'image actuelle et de l'image précédente,** la différence de temps entre ces deux peut être utilisée pour déterminer combien doit-on se déplacer lorsqu'une certaine touche est enfoncée (pour se déplacer à un vitesse constante, peu importe combien d temps prend le calcul des trames), et pour le compteur FPS.

A savoir que, **le rapport entre la longueur de la direction et le plan de la caméra détermine le FOV**. Ici, le vecteur de direction est un peu plus long que le plan de la caméra, de fait le FOV sera inférieur à 90°. Plus précisément :

$$\text{FOV} = 2^\circ \tan(= 0.66/1?0) = 66^\circ$$

\*Ce qui est parfait pour un jeu de tir à la première personne

Plus tard, **lors d'une rotation avec les touches d'entrée, les valeurs de dir et de plan seront modifiées**, mais resteront **toujours perpendiculaires et garderont la même longueur**.

## c. La fonction : Création de l'écran et calculs généraux

**Le reste de la fonction principal démarre maintenant :**

```
screen(screenWidth, screenHeight, 0, "Raycaster");
```

Tout d'abord, l'écran est créé avec une résolution de choix. Plus la résolution est grande (ex : 1280 \* 1024) plus l'effet sera lent, non pas parce que l'algorithme de raycasting est lent, mais plus simplement parce que le téléchargement d'un écran entier du CPU (Central Processing Unit, désigne la plupart du temps le processeur d'un ordinateur) vers la carte vidéo est lent.

Après avoir configuré l'écran, la gameloop démarre, c'est-à-dire la boucle qui dessine une image entière et lit l'entrée à chaque fois :

```
while(!done())
{
```

A présent, c'est ici que commence la diffusion de rayons réels. La boucle de raycasting est une boucle for qui traverse tous les x, il n'y a donc pas de calcul pour chaque pixel de l'écran, mais seulement pour chaque bande verticale :

```
for(int x = 0; x < w; x++)
{
    //calculate ray position and direction
    double cameraX = 2 * x / double(w) - 1; //x-coordinate in camera space
    double rayDirX = dirX + planeX * cameraX;
    double rayDirY = dirY + planeY * cameraX;
```

La boucle de diffusion commence par calculer ou supprimer certains variable :

- Le rayon commence à la position du joueur (posX, posY);
- camerax est la coordonnée x actuelle de l'écran (effectuée de ce cette façon afin que le côté droit de l'écran obtient la coordonnée 1, le centre de l'écran obtient la coordonnée 0 et le côté gauche de l'écran obtient la coordonnée -1);
- De là, la direction du rayon peut être calculée comme expliqué précédemment (= la somme du vecteur de direction + une partie du vecteur plan). Cela doit être fait à la fois pour les coordonnées x et y du vecteur puisque l'ajout de deux vecteurs consiste à ajouter leurs coordonnées x et à ajouter leurs coordonnées y.

---

#### d. Calculs : les calculs nécessaires pour l'algorithme ADN :

Dans le morceau de code suivant, plus de variables sont déclarées et calculées, celles-ci sont pertinentes pour l'algorithme ADN (Analyseur Différentiel Numérique, DDA pour *Digital Differential Analyzer* en anglais, est un algorithme qui permet de tracer des approximations de segments de droites sur des média discrets) :

```

//which box of the map we're in
int mapX = int(posX);
int mapY = int(posY);

//length of ray from current position to next x or y-side
double sideDistX;
double sideDistY;

//length of ray from one x or y-side to next x or y-side
double deltaDistX = std::abs(1 / rayDirX);
double deltaDistY = std::abs(1 / rayDirY);
double perpWallDist;

//what direction to step in x or y-direction (either +1 or -1)
int stepX;
int stepY;

int hit = 0; //was there a wall hit?
int side; //was a NS or a EW wall hit?

```

- **MapX et mapY représentent le carré actuel de la carte dans laquelle se trouve le rayon.** La position du rayon lui-même est un **nombre à virgule flottante** et qui contient à la fois des informations sur le **carré de la carte où l'on se trouve et où l'on se trouve dans ce carré**, mais mapX et mapY ne sont que les coordonnées de ce carré.
- **sideDistX et sideDistY sont initialement la distance que le rayon doit parcourir entre sa position de départ et le premier côté x et le premier côté y** (plus tard dans le code, leur signification changera légèrement).
- **deltaDistX et deltaDistY sont la distance que le rayon doit parcourir pour aller d'un côté x au côté x suivant, ou d'un côté y au côté y suivant.**
- **La variable perpWallDist sera utilisée plus tard pour calculer la longueur du rayon.**
- **L'algorithme DNA sautera toujours exactement d'un carré par boucle, soit un carré dans la direction x, soit un carré dans la direction y.** Si l'algorithme doit aller dans la direction x négative ou positive, et la direction y négative ou positive, cela dépendra de la direction du rayon, et **cette action sera stockée dans les variables stepX et stepY** (ces variables sont toujours de -1 ou de + 1).
- **hit est utilisé pour déterminer si la boucle à venir peut être terminée ou non.**
- **La valeur de side dépendra de si une côté x ou un côté y d'un mur a été touché :** si un côté x a été touché side est mis à 0, si un côté y a été touché side est mis à 1 (on entend par côté x et côté y les lignes de la grilles qui sont les frontières entre deux carrés).

L'image suivante montre le sideDist X initial, sideDistY et deltaDistX et deltaDistY :

**Lors de la dérivation géométrique de deltaDistX, on obtient avec Pythagores, les formules suivantes :**

$$\begin{aligned} \text{deltaDistX} &= \sqrt{(1 + (\text{rayDirY} * \text{rayDirY}) / (\text{rayDirX} * \text{rayDirX}))} \\ \text{deltaDistY} &= \sqrt{(1 + (\text{rayDirX} * \text{rayDirX}) / (\text{rayDirY} * \text{rayDirY}))} \end{aligned}$$

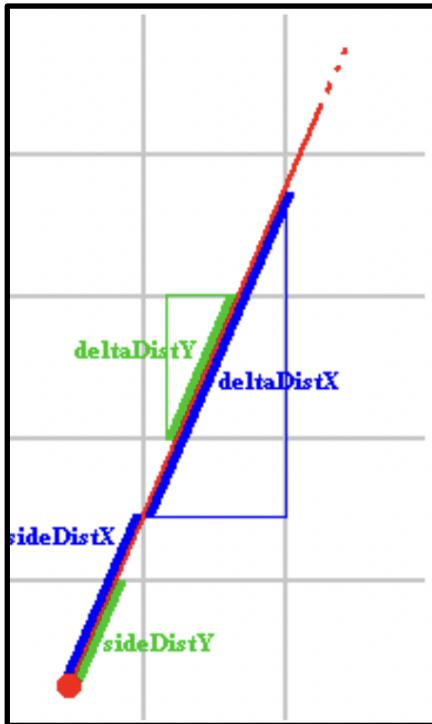
Qui peuvent être **simplifiées** de la manière suivante :

```

deltaDistX = abs (| v | / rayDirX)
deltaDistY = abs (| v | / rayDirY)

```

\*Où  $| v |$  est la longueur du vecteur rayDirX, rayDirY (c'est-à-dire  $(rayDirX * rayDirX + rayDirY * rayDirY)$ )



Cependant, l'on peut utiliser **1 au lieu de  $| v |$** , car seul le \* ratio \* entre deltaDistX et deltaDistY compte pour le code DNA, ce qui nous donne :

```

deltaDistX = abs (1 / rayDirX)
deltaDistY = abs (1 / rayDirY)

```

**Attention : Si rayDirX ou rayDirY sont à 0, la division par 0 dans le calcul ci-dessus, rend la valeur de deltaDist X ou de deltaDistY infinie.**

Si le système utilise la norme à virgule flottante IEE 754, alors le système ne lève pas d'exceptions pour cela : l'infini sera utilisé correction dans la comparaison dans les étapes DNA. Cependant, si on utilise un langage de programmation qui ne le permet pas, l'on peut utiliser ce qui suit, ce qui permettra à la boucle DDA de fonctionner également correctement en définissant à la place la valeur finie sur 0 :

```

// Alternative code for deltaDist in case division through zero is not supported
double deltaDistX = (rayDirY == 0) ? 0 : ((rayDirX == 0) ? 1 : abs(1 / rayDirX));
double deltaDistY = (rayDirX == 0) ? 0 : ((rayDirY == 0) ? 1 : abs(1 / rayDirY));

```

#### e. Calculs : Dernier calculs à faire avant le réel DNA

Maintenant, avant que le DNA réel puisse démarrer, le premier stepX, setpY et le sideDistX initial et sideDistY doivent encore être calculés :

```
//calculate step and initial sideDist
if (rayDirX < 0)
{
    stepX = -1;
    sideDistX = (posX - mapX) * deltaDistX;
}
else
{
    stepX = 1;
    sideDistX = (mapX + 1.0 - posX) * deltaDistX;
}
if (rayDirY < 0)
{
    stepY = -1;
    sideDistY = (posY - mapY) * deltaDistY;
}
else
{
    stepY = 1;
    sideDistY = (mapY + 1.0 - posY) * deltaDistY;
}
```

Si la direction du rayon à une composante x négative, stepX vaut -1. Si la direction du rayon a une composante x positive, stepX vaut +1. Si le composant x est 0, la valeur de stepX importe peu puisqu'il restera inutilisé. Il en va de même pour la composante y.

Si la direction du rayon a une composante x negative, sideDistX est la distance entre la position de départ du rayon et le premier côté vers la gauche. Si la direction du rayon a une composante x positive, le premier côté vers la droite est utilisé à la place.

Si la direction du rayon a une composante y negative, sideDistY est la distance entre la position de départ du rayon et le premier côté au-dessus de la position. Si la direction du rayon a une composante y positive, le premier côté en dessous est utilisé à la place.

Pour ces valeurs, la valeur entière mapX est utilisée et la position réelle en est soustraite, et 1.0 est ajouté dans certains cas selon si on utilise le côté à gauche ou à droite, ou si le haut ou le bas est utilisé.

Ainsi, on obtient la distance perpendiculaire à ce côté. Pour obtenir la distance euclidienne réelle : on multiplie la distance perpendiculaire à ce côté avec deltaDistX ou deltaDistY.

---

f. **Début de l'algorithme Analyseur Différentiel Numérique (DNA) :**

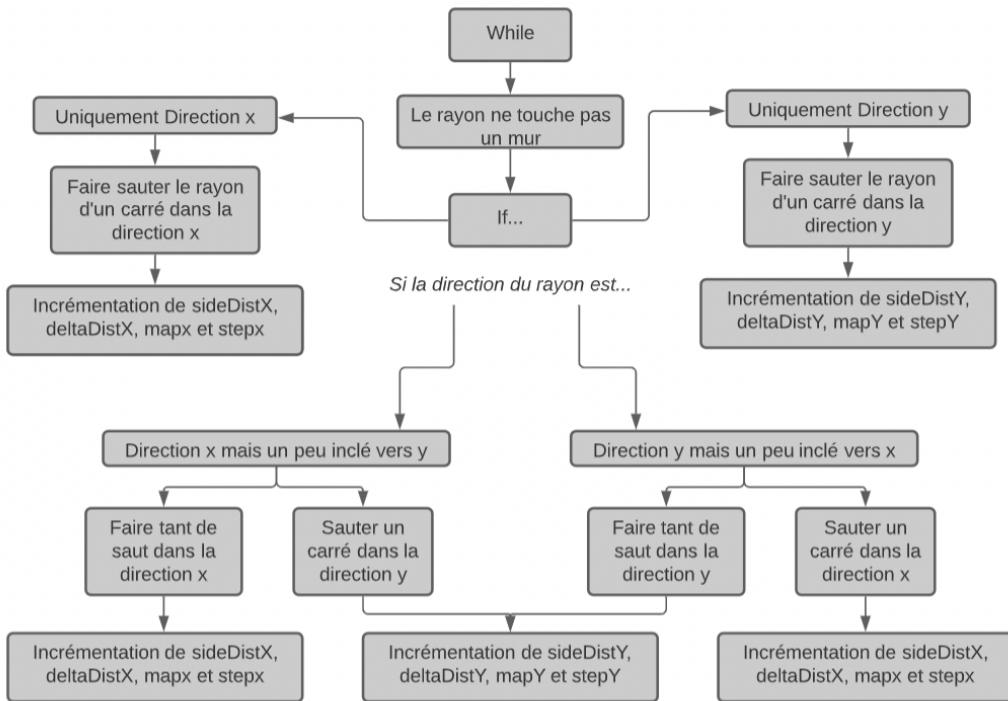
Maintenant, le DNA proprement dit démarre.

```
//perform DDA
while (hit == 0)
{
    //jump to next map square, OR in x-direction, OR in y-direction
    if (sideDistX < sideDistY)
    {
        sideDistX += deltaDistX;
        mapX += stepX;
        side = 0;
    }
    else
    {
        sideDistY += deltaDistY;
        mapY += stepY;
        side = 1;
    }
    //Check if ray has hit a wall
    if (worldMap[mapX][mapY] > 0) hit = 1;
}
```

En théorie, c'est une boucle qui incrémente le rayon de 1 carré à chaque fois, jusqu'à ce qu'un mur soit touché. Chaque fois qu'il saute un carré dans la direction x (avec stepX) ou un carré dans la direction y (avec stepY), il saute toujours 1 carré à la fois. Si la direction du rayon était la direction x, la boucle n'aura qu'à sauter d'un carré dans la direction x à chaque, car le rayon ne changera jamais sa direction y. Si le rayon est un peu incliné vers la direction y, alors, tous les de saut la direction x, le rayon devra sauter d'un carré dans la direction y. Si le rayon est exactement dans la direction y, il n'a jamais à sauter dans direction x, etc.

**sideDistX et sideDistY sont incrémentés avec deltaDistX ou deltaDistY à chaque saut dans leur direction, et mapX et mapY sont incrémentés respectivement avec stepX et stepY.**

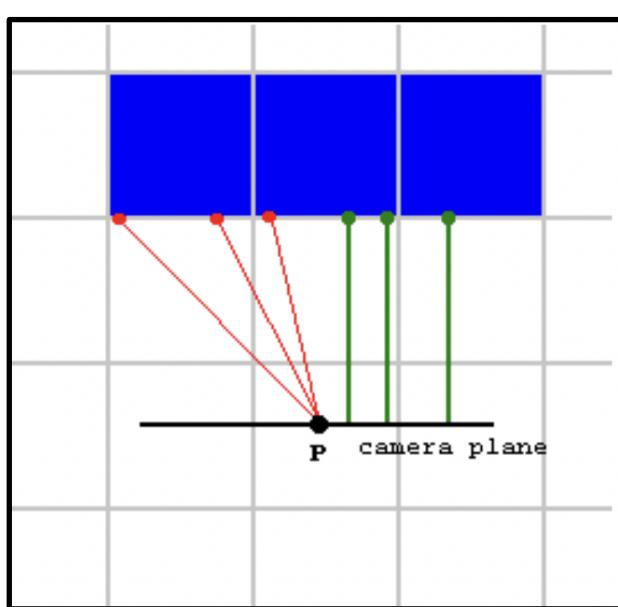
Une fois que le rayon hert un mur, la boucle se termine, et alors on saura si un côté x ou y d'un mur à été touché dans la variable side, et quel mur a été touché avec mapX et mapY. On ne saura pas exactement où se le mur à été touché mais ce n'est pas nécessaire dans ce cas car nous n'utilisons pas de murs texturés pour le moment.



#### **g. Après le DNA : Calcul de la distance du rayon au mur (perpWallDist) :**

Une fois le DNA terminé, il faut **calculer la distance du rayon au mur**, afin de pouvoir calculer à quelle hauteur le mur doit être dessiné après cela.

Ici, on n'utilise pas la distance euclidienne au point représentant le joueur, mais plutôt **la distance au plan de la caméra** (ou, la distance du point projeté sur la direction de la caméra vers le joueur), pour **éviter l'effet fisheye** (effet que l'on obtient si l'on utilise la distance réelle et qui rend tous les murs arrondis).



#### Explications avec Image :

Avec **P** le joueur, et la **ligne noir** le **plan de la caméra** on trouve :

A gauche du joueur, les **rayons rouges** **représentent la distance euclidienne** : ici, les rayons rouges vont du point de vue sur le mur au joueur.

A droite, les **rayons vert** la **distance perpendiculaire** : ici, les rayons verts vont du point de vue sur le mur directement au plan de la caméra (au lieu du joueur).

Dans cette image, le joueur regarde directement le mur, et dans ce cas, on s'attend à ce que le bas et le haut du mur forment une ligne parfaitement horizontale

sur l'écran. Cependant, les rayons rouges ont tous une longueur différente, donc il faudrait calculer différentes hauteurs de mur pour différentes bandes verticales, d'où l'effet fisheye. Les rayons vers sur la droite ont tous la même longueur, donc un même calcul avec un même résultat correct.

La même chose s'applique quand le joueur tourne : le plan de la caméra n'est plus horizontal et les lignes vertes auront des longueurs différentes, mais toujours avec un changement constant entre chacune et les murs deviennent des lignes diagonales mais droites sur l'écran.

```
//Calculate distance projected on camera direction (Euclidean distance will give fisheye effect!)
if (side == 0) perpWallDist = (mapX - posX + (1 - stepX) / 2) / rayDirX;
else           perpWallDist = (mapY - posY + (1 - stepY) / 2) / rayDirY;
```

Dans le code ci-dessus, **(1-stepX) / 2 vaut 1 si stepX = -1 et 0 si stepX = 1**. Cela est nécessaire dans le sous où il faut ajouter 1 à la longueur lorsque  $\text{rayDirX} < 0$ , c'est pour la même raison que 1.0 a été ajouté à la valeur initiale de  $\text{sideDistX}$  dans un cas mais pas dans l'autre.

**La distance est alors calculée :**

**Si un côté X est touché alors  $\text{mapX}-\text{posX} + (1-\text{stepX}) / 2$  = nombre de carrés que le rayon a traversé dans la direction X** (ce n'est pas forcément un nombre entier).

**Si le rayon est perpendiculaire au côté X**, c'est déjà la **valeur correcte**, mais comme la direction du rayon est différente la plupart du temps, sa distance perpendiculaire réelle sera plus grande, **il faudra donc la diviser par la coordonnée X du rayDir vecteur**.

\*Le calcul est similaire dans le où une côté y est atteint.  
\*La distance calculée n'est jamais négative.

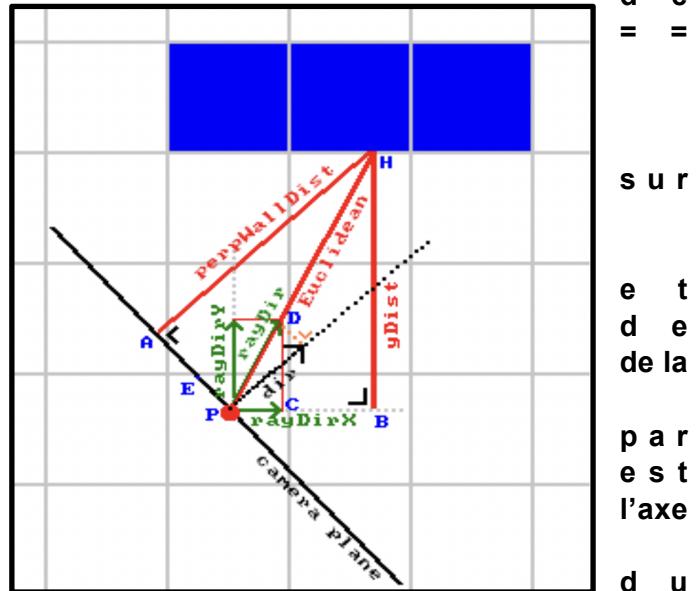
#### h. Une autre façon de calculer distance du rayon au mur (perpWallDist) :

Une autre façon de calculer perpWallDist est d'utiliser la formule de distance d'un point à l'autre en 2D, en utilisant le point de repère du mur et le plan de la caméra du joueur. Cependant, cette méthode est plus coûteuse en calculs.

#### Explications :

La situation est représentée en vue dessus en 2D, pour le cas y (côté == 0 est la même) :

- P : position du joueur
- H : point de vie du rayon le mur.
- A : point du plan de la caméra le plus proche de H est le point d'intersection perpWallDist avec le plan caméra.
- B : point de l'axe X passant le lecteur le proche de H et le point où yDist croise X passant par le lecteur.
- C : pointez sur la position joueur + rayDirX.
- D : pointer sur la position du joueur + rayDir.
- E : le point D avec le vecteur dir soustrait ( $E + \text{dir} = D$ );
- Les points A, B, C, D, E, H et P forment des triangles qui sont considérés: BHP, CDP, AHP et DEP.
- **perpWallDist** : la longueur de cette ligne est la valeur à calculer maintenant, soit la distance perpendiculaire du point de frappe du mur au plan de la caméra au lieu de la distance euclidienne au point du joueur.
- **yDist** correspond à ce qui est " $(\text{mapY} - \text{posY} (1 - \text{stepY}) / 2)$ " dans le code ci-dessus : c'est la coordonnées y du vecteur de distance euclidienne, en coordonnées mondiales.
- **Euclidienne** est la distance euclidienne entre le joueur P et le point de frappe exact H : sa direction est le rayDir, mais sa longueur va jusqu'au mur.
- **rayDir** est la direction du rayon marqué "Euclidienne" correspondant aux variables **rayDirX** et **rayDirY** dans le code (à noter que sa longueur  $| \text{rayDir} |$  n'est pas 1 mais légèrement plus élevé, en raison de la façon dont on a ajouté une valeur à  $\text{dirX}$  et  $\text{dirY}$  (le vecteur dir, qui est normalisé à 1) dans le code).
- **rayDirX** et **rayDirY** : composants X et Y de **rayDir** correspondant aux variable **rayDirX** et **rayDirY** dans le code.
- **dir** : la direction de recherche du joueur principale, donné par **dirX** et **dirY** dans le code (la longueur de ce vecteur est toujours exactement 1, ce qui correspond à la direction de recherche au centre de l'écran, par opposition à la direction du rayon actuel) : il est perpendiculaire au plan de la caméra et **perpWallDist** est parallèle à celui-là.
- La ligne pointillée orange : la valeur qui a été ajoutée à **dir** pour obtenir **rayDir** et est parallèle au plan de la caméra et perpendiculaire à **dir**.
- **plan de la caméra** : le plan de la caméra, la ligne donnée par **cameraX** et **cameraY**, perpendiculaire à la direction de regard du joueur principal.



Et la dérivation du calcul `peprWallDist` est alors :

- “`(ampY - posY + (1 - stepY) / 2) / rayDir`” est “`yDist / rayDirY`” dans l'image.
  - Les triangles PHB et PCD ont la même forme mais de tailles différentes, de fait ils ont le même rapport d'arêtes, il en est de même pour les triangles AHP et EDP (ici, la longueur de l'arête ED, c'est-à-dire  $|ED|$ , est égale à la longueur de dir,  $|dir|$ , qui vaut 1, de même pour  $|PD|$  qui est égale à  $|rayDir|$ ).
  - Les triangles démontrent que le rapport `yDist / rayDistY` est égale au rapport Euclidien /  $|rayDir|$ . Ainsi, maintenant on peut dériver `perpWakkDist = Euclidean / |rayDir|`.
  - Les triangles démontrent le rapport suivant : Euclidien /  $|rayDir|$  = `perpWallDist / |dir|` = `perpWallDist / 1`.
  - Enfin, toutes ces étapes montrent que : `perpWallDist = yDist / rayDirY`.
- 

#### i. Calcul de la hauteur en pixel de l'écran :

Maintenant que nous avons calculé la distance (`perpWallDist`), il faut **calculer la hauteur de la ligne qui doit être dessinée à l'écran** (c'est l'inverse de `perpWallDist`), puis **multipliée par h** (la hauteur en pixels), afin de **déterminer les coordonnées des pixels**. Bien évidemment, on peut multiplier par une autre valeur comme  $2 * h$  si l'on souhaite obtenir des plus hauts ou plus bas. **La valeur de h va nous permettre donner au mur une ressemblance à des cubes de hauteur, largeur et profondeurs égales.**

Ensuite, à partir de cette `lineHeight` (la hauteur de la ligne verticales à tracer), on va **calculer les positions de début et de fin de l'endroit qu'on doit dessiner, où le centre du mur doit être au centre de l'écran** (si ces points se trouvent à l'extérieur de l'écran, ils seront plafonnées à 0 ou  $h - 1$ ).

```
//Calculate height of line to draw on screen
int lineHeight = (int)(h / perpWallDist);

//calculate lowest and highest pixel to fill in current
int drawStart = -lineHeight / 2 + h / 2;
if(drawStart < 0)drawStart = 0;
int drawEnd = lineHeight / 2 + h / 2;
if(drawEnd >= h)drawEnd = h - 1;
```

---

j. Déterminer la couleur de mur :

Enfin, en fonction du numéro de mur touché, une couleur est choisie. Si un côté a été touché, alors la couleur est assombrie, ce qui donnera un effet plus agréable. La ligne verticale est alors dessinée avec la commande verLine : cela marque la fin de la boucle de diffusion de rayons (après avoir fait cela pour au moins chaque x bien évidemment).

```
//choose wall color
ColorRGB color;
switch(worldMap[mapX][mapY])
{
    case 1: color = RGB_Red; break; //red
    case 2: color = RGB_Green; break; //green
    case 3: color = RGB_Blue; break; //blue
    case 4: color = RGB_White; break; //white
    default: color = RGB_Yellow; break; //yellow
}

//give x and y sides different brightness
if (side == 1) {color = color / 2;}

//draw the pixels of the stripe as a vertical line
verLine(x, drawStart, drawEnd, color);
}
```

#### **k. Derniers calculs :**

Une fois la boucle de diffusion de rayons terminées, l'**heure de l'image actuelle et de l'image précédente sont calculées**, le **FPS** (images par seconde) est également calculé et imprimé et l'écran est redessiné pour que tout (tous les murs, et la valeur des FPS) deviennent visible.

Après cela, le backbuffer est effacé avec **cls()**, de sorte que lorsqu'on dessine à nouveau les murs du cadre suivant, le sol et le plafond redeviendront noirs au lieu de contenir encore des pixels de l'image précédente.

Les modificateurs de vitesse utilisent **frameTime** et une valeur constante pour déterminer la vitesse de déplacement et de rotation des touches d'entrée. Grâce à l'utilisation de **frameTime**, l'on peut s'assurer que la vitesse de déplacement et de rotation est indépendante à la vitesse du processeur.

```
//timing for input and FPS counter
oldTime = time;
time = getTicks();
double frameTime = (time - oldTime) / 1000.0; //frameTime is the time this frame has taken, in seconds
print(1.0 / frameTime); //FPS counter
redraw();
cls();

//speed modifiers
double moveSpeed = frameTime * 5.0; //the constant value is in squares/second
double rotSpeed = frameTime * 3.0; //the constant value is in radians/second
```

## I. Lecture des clés :

La dernière partie est la **partie d'entrée où les clés sont lues**. Si l'on appuie sur la flèche du haut, le joueur avancera : de fait il faudra ajouter dirX à posX et dirY à posY. Cela suppose que dirX et dirY sont des vecteurs normalisés (leur longueur est de 1, valeur qui leurs ont été initialement définis).

De plus, il faut aussi s'assurer de la **détection de collision intégrée** : à savoir si la nouvelle position sera à l'intérieur d'un mur, dans ce cas là on ne bougera pas. Cette détection de collision peut être améliorée en vérifiant si un cercle autour du joueur n'ira pas à l'intérieur du mur au lieu d'un seul point par exemple.

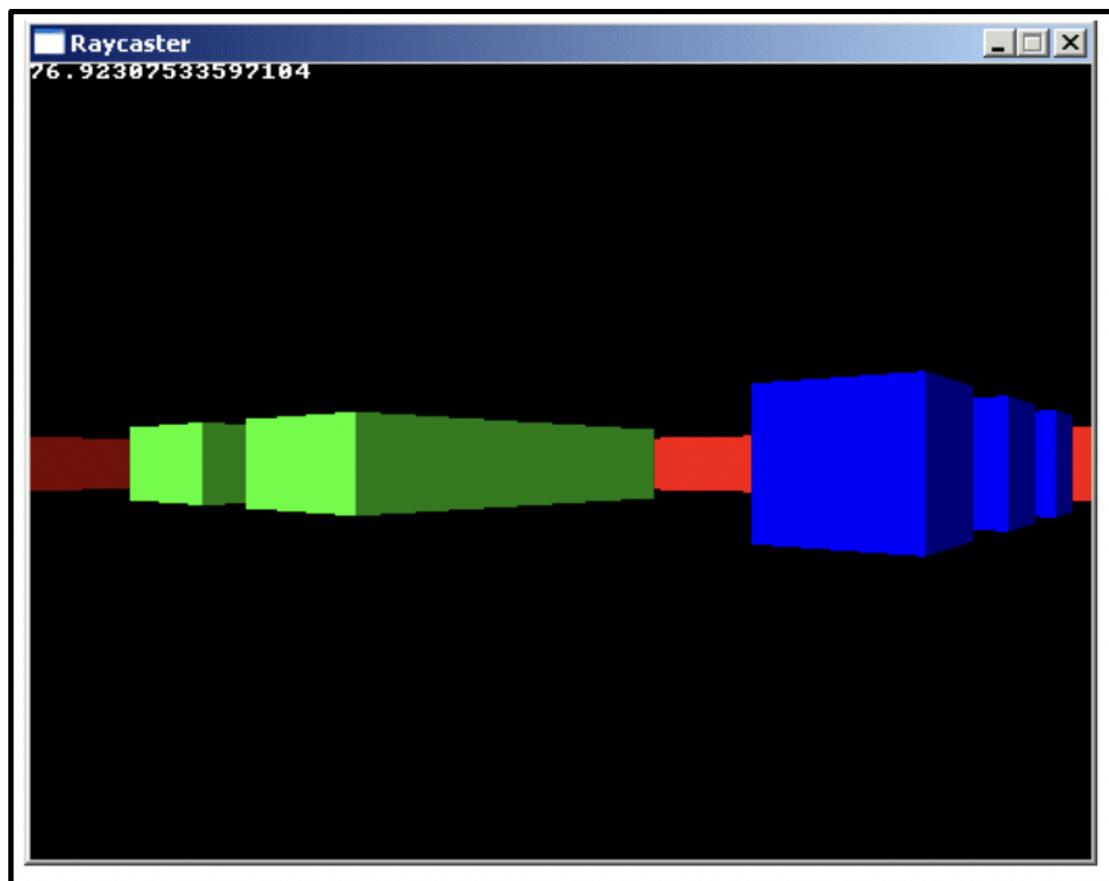
**La même chose est faite si l'on appuie sur la flèche du bas, mais ici la direction est soustraire à la place.**

**Pour faire pivoter, si la flèche gauche ou droite est enfoncee, le vecteur de direction et le vecteur de plan sont tournées en utilisant les formules de multiplication avec la matrice de rotation** (et sur l'angle rotSpeed).

```
readKeys();
//move forward if no wall in front of you
if (keyDown(SDLK_UP))
{
    if(worldMap[int(posX + dirX * moveSpeed)][int(posY)] == false) posX += dirX * moveSpeed;
    if(worldMap[int(posX)][int(posY + dirY * moveSpeed)] == false) posY += dirY * moveSpeed;
}
//move backwards if no wall behind you
if (keyDown(SDLK_DOWN))
{
    if(worldMap[int(posX - dirX * moveSpeed)][int(posY)] == false) posX -= dirX * moveSpeed;
    if(worldMap[int(posX)][int(posY - dirY * moveSpeed)] == false) posY -= dirY * moveSpeed;
}
//rotate to the right
if (keyDown(SDLK_RIGHT))
{
    //both camera direction and camera plane must be rotated
    double oldDirX = dirX;
    dirX = dirX * cos(-rotSpeed) - dirY * sin(-rotSpeed);
    dirY = oldDirX * sin(-rotSpeed) + dirY * cos(-rotSpeed);
    double oldPlaneX = planeX;
    planeX = planeX * cos(-rotSpeed) - planeY * sin(-rotSpeed);
    planeY = oldPlaneX * sin(-rotSpeed) + planeY * cos(-rotSpeed);
}
//rotate to the left
if (keyDown(SDLK_LEFT))
{
    //both camera direction and camera plane must be rotated
    double oldDirX = dirX;
    dirX = dirX * cos(rotSpeed) - dirY * sin(rotSpeed);
    dirY = oldDirX * sin(rotSpeed) + dirY * cos(rotSpeed);
    double oldPlaneX = planeX;
    planeX = planeX * cos(rotSpeed) - planeY * sin(rotSpeed);
    planeY = oldPlaneX * sin(rotSpeed) + planeY * cos(rotSpeed);
}
```

## m. Résultat :

Cela conclut le code du raycaster non texturé :



#### 4. Textured Raycasting (example) :

---

##### a. Différences avec le untextured raycasting :

Le noyau de la version texturée du raycaster est presque le même, seulement **à la fin des calculs supplémentaires doivent être effectués pour les textures**, et une boucle dans la direction y est nécessaire pour parcourir chaque pixel afin de déterminer quel **texel** (texture pixel) de la texture doit être utilisé pour cela.

**Les bandes verticales ne peuvent pas être dessinées avec la commande de ligne verticale, mais chaque pixel doit être dessiné séparément.** La meilleure façon est d'utiliser un tableau 2D comme tampon d'écran cette fois, et de le copier à l'écran à la fois (cela va beaucoup plus vite que d'utiliser pset).

Bien évidemment, on n'aura besoin d'un tableau supplémentaire pour les textures, et comme la fonction “drawbuffer” fonctionne avec des valeurs entières uniques pour les couleurs (au lieu de 3 octets séparés pour R, V et B). De fait, les textures sont également stockées dans ce format.

---

##### b. La carte du monde :

Le code est essentiellement le même que celui du “untextured raycasting” vu précédemment, sauf à quelques exceptions (les parties en gras du code sont nouvelles) :

- **screanWidth** et **screenHeight** sont maintenant définis au début car on va avoir besoin des mêmes valeurs tant pour la fonction screen que pour créer le buffer d'écran.
- **La largeur et la hauteur de la texture** sont également au début du code, elles représentent la largeur et la hauteur en texels des textures.
- **La carte du monde** est également modifiée : il s'agit d'une carte plus complexe avec des couloirs et des pièces pour montrer les différentes textures (encore une fois, les 0 sont des espaces vides pouvant être parcourus à pied, et chaque nombre positif correspond à une texture différente).

Le tampon d'écran et les tableaux de textures sont déclarés ici. Le tableau de texture est un tableau de std :: vecteurs, chacun avec une certaine largeur \* hauteur de pixels.

### c. Tableaux des textures :

Le tampon d'écran et les tableaux de textures sont déclarés ici :

```
int main(int /*argc*/, char /*argv*/[])
{
    double posX = 22.0, posY = 11.5; //x and y start position
    double dirX = -1.0, dirY = 0.0; //initial direction vector
    double planeX = 0.0, planeY = 0.66; //the 2d raycaster version of camera plane

    double time = 0; //time of current frame
    double oldTime = 0; //time of previous frame

    Uint32 buffer[screenHeight][screenWidth]; // y-coordinate first because it works per scanline
    std::vector<texture> texture[8];
    for(int i = 0; i < 8; i++) texture[i].resize(texWidth * texHeight);
```

Le tableau de texture est un tableau de std :: vecteurs, chacun avec une certaine largeur \* hauteur de pixels.

---

### d. Fonction principal :

La fonction principale commence, à présent, par générer les textures :

```
screen(screenWidth, screenHeight, 0, "Raycaster");

//generate some textures
for(int x = 0; x < texWidth; x++)
for(int y = 0; y < texHeight; y++)
{
    int xorcolor = (x * 256 / texWidth) ^ (y * 256 / texHeight);
    //int xcolor = x * 256 / texWidth;
    int ycolor = y * 256 / texHeight;
    int xycolor = y * 128 / texHeight + x * 128 / texWidth;
    texture[0][texWidth * y + x] = 65536 * 254 * (x != y && x != texWidth - y); //flat red texture with black cross
    texture[1][texWidth * y + x] = xycolor + 256 * xycolor + 65536 * xycolor; //sloped greyscale
    texture[2][texWidth * y + x] = 256 * xycolor + 65536 * xycolor; //sloped yellow gradient
    texture[3][texWidth * y + x] = xorcolor + 256 * xorcolor + 65536 * xorcolor; //xor greyscale
    texture[4][texWidth * y + x] = 256 * xorcolor; //xor green
    texture[5][texWidth * y + x] = 65536 * 192 * (x % 16 && y % 16); //red bricks
    texture[6][texWidth * y + x] = 65536 * ycolor; //red gradient
    texture[7][texWidth * y + x] = 128 + 256 * 128 + 65536 * 128; //flat grey texture
}
```

On a une double boucle qui traverse chaque pixel des textures, puis le pixel correspondant de chaque texture obtient une certaine valeur calculée à partir de x et y.

Certains textures ont un **motif OXR**, d'autres un **simple dégradé**, d'autre une sorte de **motif de brique** : fondamentalement, ce sont tous des **motifs assez simples**.

---

### e. Avant l'algorithme DNA : gameloop, déclarations et calculs

Ceci est à nouveau le début du gameloop, des déclarations et calculs initiaux avant l'algorithme DNA :

```

//start the main loop
while(!done())
{
    for(int x = 0; x < w; x++)
    {
        //calculate ray position and direction
        double cameraX = 2*x/double(w)-1; //x-coordinate in camera space
        double rayDirX = dirX + planeX*cameraX;
        double rayDirY = dirY + planeY*cameraX;

        //which box of the map we're in
        int mapX = int(posX);
        int mapY = int(posY);

        //length of ray from current position to next x or y-side
        double sideDistX;
        double sideDistY;

        //length of ray from one x or y-side to next x or y-side
        double deltaDistX = sqrt(1 + (rayDirY * rayDirY) / (rayDirX * rayDirX));
        double deltaDistY = sqrt(1 + (rayDirX * rayDirX) / (rayDirY * rayDirY));
        double perpWallDist;

        //what direction to step in x or y-direction (either +1 or -1)
        int stepX;
        int stepY;

        int hit = 0; //was there a wall hit?
        int side; //was a NS or a EW wall hit?

        //calculate step and initial sideDist
        if (rayDirX < 0)
        {
            stepX = -1;
            sideDistX = (posX - mapX) * deltaDistX;
        }
        else
        {
            stepX = 1;
            sideDistX = (mapX + 1.0 - posX) * deltaDistX;
        }
        if (rayDirY < 0)
        {
            stepY = -1;
            sideDistY = (posY - mapY) * deltaDistY;
        }
        else
        {
            stepY = 1;
            sideDistY = (mapY + 1.0 - posY) * deltaDistY;
        }
    }
}

```

## f. DDA Loop : calculs de distance et de hauteur

Ceci est encore la boucle DNA, les calculs de distances et de hauteur :

```
//perform DDA
while (hit == 0)
{
    //jump to next map square, OR in x-direction, OR in y-direction
    if (sideDistX < sideDistY)
    {
        sideDistX += deltaDistX;
        mapX += stepX;
        side = 0;
    }
    else
    {
        sideDistY += deltaDistY;
        mapY += stepY;
        side = 1;
    }
    //Check if ray has hit a wall
    if (worldMap[mapX][mapY] > 0) hit = 1;
}

//Calculate distance of perpendicular ray (Euclidean distance will give fisheye effect!)
if (side == 0) perpWallDist = (mapX - posX + (1 - stepX) / 2) / rayDirX;
else           perpWallDist = (mapY - posY + (1 - stepY) / 2) / rayDirY;

//Calculate height of line to draw on screen
int lineHeight = (int)(h / perpWallDist);

//calculate lowest and highest pixel to fill in current stripe
int drawStart = -lineHeight / 2 + h / 2;
if(drawStart < 0) drawStart = 0;
int drawEnd = lineHeight / 2 + h / 2;
if(drawEnd >= h) drawEnd = h - 1;
```

---

## g. Nouveaux calculs : Direction x

Les calculs suivant sont eux nouveaux et remplacent le sélecteur de couleur du raycaster non texturé :

```
//texturing calculations
int texNum = worldMap[mapX][mapY] - 1; //1 subtracted from it so that texture 0 can be used!

//calculate value of wallX
double wallX; //where exactly the wall was hit
if (side == 0) wallX = posY + perpWallDist * rayDirY;
else           wallX = posX + perpWallDist * rayDirX;
wallX -= floor((wallX));

//x coordinate on the texture
int texX = int(wallX * double(texWidth));
if(side == 0 && rayDirX > 0) texX = texWidth - texX - 1;
if(side == 1 && rayDirY < 0) texX = texWidth - texX - 1;
```

- **La variable texNum est la valeur du carré de la carte actuelle moins 1** (1 car il existe une texture 0). Cependant, **la tuile de carte 0 n'a pas de texture** car elle représente un espace vide. Pour pouvoir tout de même utiliser la texture 0, il est conseillé de toujours soustraire 1 pour que les tuiles de carte avec la valeur donnent la texture 0, etc.
  - **La valeur wallX représente la valeur exacte à laquelle le mur a été touché** et pas seulement les coordonnées entières du mur. Ceci est nécessaire pour savoir quelle coordonnée x de la texture on doit utiliser. Pour cela, on **calcule d'abord la coordonnée exacte de x ou y dans le monde, puis on soustrait la valeur entière du mur**. A noter que même si elle s'appelle wallX, c'est en fait une coordonnée y du mur si side est égale à 1, mais c'est toujours la coordonnée x de la texture.
  - **texX est la coordonnée x de la texture et elle est calculée à partir de wallX.**
- 

#### **h. Nouveaux calculs : Direction y**

Maintenant que l'on **connaît la coordonnée x de la texture, on sait que cette coordonnée restera la même**, car on reste dans la même bande verticale de l'écran.

A présent, l'on a besoin d'une boucle dans la direction y pour donner à chaque pixel de la bande verticale la coordonnée y correcte de la texture, appelée texY:

```
// How much to increase the texture coordinate per screen pixel
double step = 1.0 * texHeight / lineHeight;
// Starting texture coordinate
double texPos = (drawStart - h / 2 + lineHeight / 2) * step;
for(int y = drawStart; y < drawEnd; y++)
{
    // Cast the texture coordinate to integer, and mask with (texHeight - 1) in case of overflow
    int texY = (int)texPos & (texHeight - 1);
    texPos += step;
    Uint32 color = texture[texNum][texHeight * texY + texX];
    //make color darker for y-sides: R, G and B byte each divided through two with a "shift" and an "and"
    if(side == 1) color = (color >> 1) & 8355711;
    buffer[y][x] = color;
}
}
```

- **La valeur de texY est calculée en augmentant d'une de pas précalculée** (ce qui est possible car elle est constante dans la bande verticale) **pour chaque pixel**.
- **La taille du pas indique de combien augmenter les coordonnées de texture** (en virgule flottante) **pour chaque pixel en coordonnées d'écran verticales**.
- Il faut **convertir la valeur en virgule flottante en entier** pour sélectionner le **pixel de texture réel**.
- Il faut **un algorithme de bresenham** (L'algorithme de tracé de segment de Bresenham est un algorithme développé par Jack.E Bresenham en mai 1962, qui détermine quels sont les points d'un plan discret qui doivent être tracés afin de former une approximation de segment de droite entre deux points donnés) **ou DNA plus rapide en entier uniquement peut être possible pour cela**.
- **Le pas à pas effectué ici est le mappage de texture affine**, ce qui signifie que l'on peut **interpoler linéairement entre deux** plutôt que de savoir calculer une division différente pour chaque pixel.
- **La couleur du pixel à dessiner est alors simplement obtenue à partir de la texture [texNum] [texX] [texY]**, qui est le texel correct de la texture correcte.

Ici, comme pour le raycaster non texturé, **on rend la valeur de la couleur plus sombre si un côté y du mur a été touché** (comme s'il y avait une sorte d'éclairage). Cependant, comme la valeur de couleur n'existe pas à partir d'une valeur R, V et B séparée, mais que ces 3 octets sont collés ensemble dans un seul entier, un calcul pas si intuitif est utilisé.

**La couleur est assombrie en divisant R, V et B par 2.** La division d'un nombre décimal par 10 peut être effectuée en supprimant le dernier chiffre (ex : 300/10 vaut 30). De même, diviser un nombre binaire par 2 revient à supprimer le dernier bit. Cela peut être fait en décalant le bit vers la droite avec `>> 1`. Mais, ici, on décale en bits un entier de 24 bits (en fait 32 bits, mais les 8 premiers bits ne sont pas utilisés). Pour cette raison, le dernier bit d'un octet deviendra le premier bit de l'octet suivant, et cela déraille les valeurs de couleur ! Ainsi, **après le décalage de bits, le premier bit de chaque octet doit être mis à zéro**, et cela peut être fait par “**AND-ing**” binaire la valeur avec la valeur binaire : **0111111011111101111111**, qui est **8355711** en décimal (le résultat est donc bien une couleur plus foncée).

Enfin, le pixel tampon actuel est défini sur cette couleur et on passe au y suivant.

---

#### i. Dessiner et effacer le tampon :

Maintenant, le tampon doit encore être dessiné, et après cela, il doit être effacé en utilisant “**cls**”. Il faut s'assurer de le faire dans l'**ordre de la ligne de balayage pour la vitesse grâce à la localité mémoire pour la mise en cache**. Sinon le reste de ce code est à nous le même :

```
drawBuffer(buffer[0]);
for(int y = 0; y < h; y++) for(int x = 0; x < w; x++) buffer[y][x] = 0; //clear the buffer instead of cls()
//timing for input and FPS counter
oldTime = time;
time = getTicks();
double frameTime = (time - oldTime) / 1000.0; //frametime is the time this frame has taken, in seconds
print(1.0 / frameTime); //FPS counter
redraw();

//speed modifiers
double moveSpeed = frameTime * 5.0; //the constant value is in squares/second
double rotSpeed = frameTime * 3.0; //the constant value is in radians/second
```

---

#### j. Lecture des clés :

Et voici encore les clés, rien n'a changé ici non plus :

```

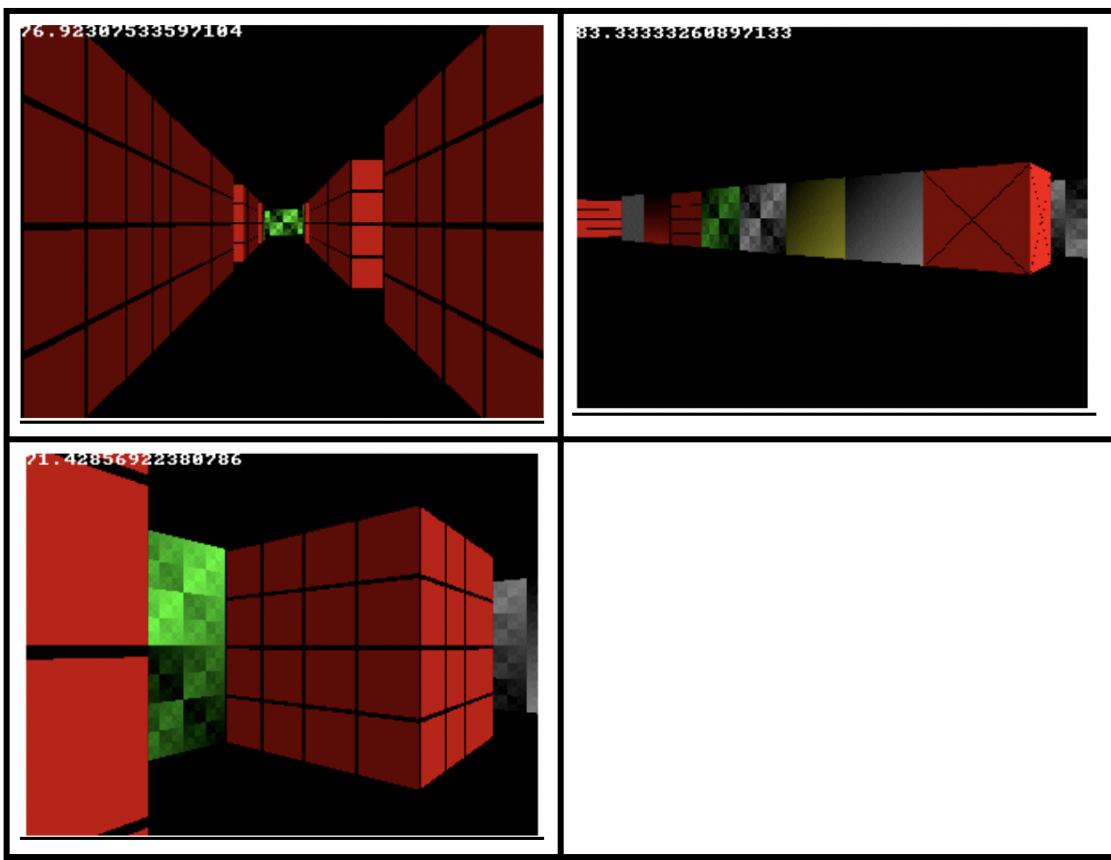
readKeys();
//move forward if no wall in front of you
if (keyDown(SDLK_UP))
{
    if(worldMap[int(posX + dirX * moveSpeed)][int(posY)] == false) posX += dirX * moveSpeed;
    if(worldMap[int(posX)][int(posY + dirY * moveSpeed)] == false) posY += dirY * moveSpeed;
}
//move backwards if no wall behind you
if (keyDown(SDLK_DOWN))
{
    if(worldMap[int(posX - dirX * moveSpeed)][int(posY)] == false) posX -= dirX * moveSpeed;
    if(worldMap[int(posX)][int(posY - dirY * moveSpeed)] == false) posY -= dirY * moveSpeed;
}
//rotate to the right
if (keyDown(SDLK_RIGHT))
{
    //both camera direction and camera plane must be rotated
    double oldDirX = dirX;
    dirX = dirX * cos(-rotSpeed) - dirY * sin(-rotSpeed);
    dirY = oldDirX * sin(-rotSpeed) + dirY * cos(-rotSpeed);
    double oldPlaneX = planeX;
    planeX = planeX * cos(-rotSpeed) - planeY * sin(-rotSpeed);
    planeY = oldPlaneX * sin(-rotSpeed) + planeY * cos(-rotSpeed);
}
//rotate to the left
if (keyDown(SDLK_LEFT))
{
    //both camera direction and camera plane must be rotated
    double oldDirX = dirX;
    dirX = dirX * cos(rotSpeed) - dirY * sin(rotSpeed);
    dirY = oldDirX * sin(rotSpeed) + dirY * cos(rotSpeed);
    double oldPlaneX = planeX;
    planeX = planeX * cos(rotSpeed) - planeY * sin(rotSpeed);
    planeY = oldPlaneX * sin(rotSpeed) + planeY * cos(rotSpeed);
}
}

```

\*Si on le souhaite, on peut essayer d'ajouter des clés à mitrailler (à mitrailler à gauche et à droite) : celles-ci doivent être faites de la même manière que les touches haut et bas, mais en utilisant planeX et planY au lieu de dirX et dirY.

---

## k. Résultat :



## I. Remarques :

Habituellement, les images sont stockées par des lignes de balayage horizontales, mais pour un raycaster, les textures sont désignées sous forme de bandes verticales. Par conséquent, pour utiliser de manière optimale le cache du processeur et éviter les échecs de page, il est peut être plus efficace de stocker les textures dans la mémoire bande verticale par bande verticale, plutôt que par ligne de balayage horizontal. Pour ce faire, après avoir générée les textures il faut permuter leurs X et Y par (ce code ne fonctionne que si texWidth et texHeight sont identiques) :

```
//swap texture X/Y since they'll be used as vertical stripes
for(size_t i = 0; i < 8; i++)
    for(size_t x = 0; x < texSize; x++)
        for(size_t y = 0; y < x; y++)
            std::swap(texture[i][texSize * y + x], texture[i][texSize * x + y]);
```

Ou on peut tout simplement **échanger X et Y là où les textures sont générées**, mais dans de nombreux cas après avoir chargé une image ou après avoir obtenu une texture à partir d'autres formats (on aura quand même dans les lignes balayage et on devra quand même l'échanger de cette façon).

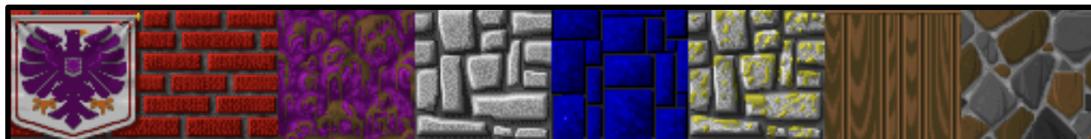
**Lorsqu' on récupère le pixel de la texture, il faudrait plutôt utiliser le code suivant :**

```
//swap texture X/Y since they'll be used as vertical stripes
for(size_t i = 0; i < 8; i++)
for(size_t x = 0; x < texSize; x++)
for(size_t y = 0; y < x; y++)
std::swap(texture[i][texSize * y + x], texture[i][texSize * x + y]);
```

---

## 5. Wolfenstein 3D Textures :

Les 8 textures suivantes proviennent de Wolfenstein 3D et sont protégés par le droit d'auteur d'ID Software :



Pour les utiliser, il suffit tout simplement de remplacer la partie du code qui génère les motifs de textures par ce qui suit (et s'assurer que ces textures sont dans le bon chemin) :

```
//generate some textures
unsigned long tw, th;
loadImage(texture[0], tw, th, "pics/eagle.png");
loadImage(texture[1], tw, th, "pics/redbrick.png");
loadImage(texture[2], tw, th, "pics/purplestone.png");
loadImage(texture[3], tw, th, "pics/greystone.png");
loadImage(texture[4], tw, th, "pics/bluestone.png");
loadImage(texture[5], tw, th, "pics/mossy.png");
loadImage(texture[6], tw, th, "pics/wood.png");
loadImage(texture[7], tw, th, "pics/colorstone.png");
```

Dans le Wolfenstein 3D original, les couleurs d'un côté ont également été rendues plus sombres que la couleur de l'autre côté d'un mur pour créer l'effet d'ombre, mais ils ont utilisé une texture séparée à chaque fois (une sombre et une claire).



## **6. Les Performances de Wolfenstein 3D :**

Sur un ordinateur moderne, lors de l'utilisation de la haute résolution (4K, à partir de 2019, **ce logiciel raycaster sera plus lent** que certains graphiques 3D beaucoup plus complexes rendus sur le GPU (processeur graphiques) avec une carte graphique 3D.

Il y a au moins **deux problèmes de ralentissement de la vitesse du code raycaster** dans ce tutoriel :

- **On ne peut pas prendre en compte si l'on souhaite créer un raycaster ultra rapide pour de très hautes résolutions.** Le raycasting fonctionne avec des bandes verticales, mais **la mémoire tampon d'écran en mémoire est disposée avec des lignes de balayage horizontales.** Ainsi, **dessiner des bandes verticales est mauvais pour la mémoire pour la mise en cache et la perte d'une bonne mise en cache peut nuire à la vitesse plus que certains des calculs 3D sur les machines modernes.**
  - **Le raycasting utilise le logiciel blitting avec SDL** (dans QuickCG, avec redraw()) qui est lent pour les grandes résolutions par rapport au matériel. L'utilisation probable de SDL par QuickCG n'est pas optimale, par exemple : l'utilisation d'OpenGL peut être plus rapide (même pour le rendu logiciel), ce qui peut être réparé en arrière-plan.
-

## **VI. Etape 5 : Raycasting II (Floor and Ceiling)**

---

### **1. Introduction :**

Précédemment, il a été montré comment rendre des murs plats non texturés et comment rendre ceux texturés. Cependant, **le sol et le plafond** sont toujours restés plats et non texturés. Pour que ces derniers soient **texturés, des calculs supplémentaires sont nécessaires.**

Wolfenstein 3D n'avait pas de texture au sol et au plafond, mais d'autres jeux de raycasting qui ont suivi peu de temps après Wolf3D les eut, comme **Blake Stone 3D**:



### **2. Comment ça marche :**

Contrairement aux textures de mur, **les textures de sol et de plafond sont horizontales et ne peuvent donc pas être dessinées de la même manière que le mur avec de rayures verticales.** Au lieu de cela, **ils sont dessinés avec des lignes de balayage horizontales.**

**La perspective est similaire à celle des murs mais pivotés à 90°**, mais contrairement aux murs qui utilisaient exactement 1 texture par bandes verticales, **plusieurs textures de sol** (ou la même à plusieurs reprises) **peuvent traverser notre ligne horizontale.**

Dessiner le plafond se déroule de la même manière que dessiner le sol, donc seul le sol va être traité.

**Le moulage du sol se fait avec les murs**, donc on dessine d'abord tout le sol (et le plafond), puis on écrase une partie des pixels avec les murs. En bref, le moulage au sol fonctionne comme suit : **travail scanline par scanline**.

- Pour la ligne de balayage actuelle, calculer la position sur le sol correspondant au pixel gauche de la ligne de balayage et la position correspondant au pixel droit. Cela peut être calculé comme l'endroit où le rayon partant de la caméra, passant par ce pixel du plan de la caméra, atteint le sol.
  - Ensuite, on peut interpoler linéairement entre ce point le plus à gauche et le point le plus à droite pour obtenir les coordonnées du sol correspondant aux autres pixels de cette ligne de balayage. Cela fonctionne car la texture du sol est parfaitement horizontale. S'il était incliné, on aurait besoin de faire un mappage de texture correct en perspective plus coûteux à la place.
-

### 3. Le Code :

#### a. Première Partie :

La première partie du code est exactement la même que dans le raycasting des murs. Il y a également une nouvelle carte. Ce morceau de code déclare toutes les variables nécessaires, charge les textures et dessine dans murales verticales texturées :

```
#define screenWidth 640
#define screenHeight 480
#define texWidth 64
#define texHeight 64
#define mapWidth 24
#define mapHeight 24

int worldMap[mapWidth][mapHeight]=
{
    {8,8,8,8,8,8,8,8,8,4,4,6,4,4,6,4,4,4,6,4},
    {8,0,0,0,0,0,0,0,0,8,4,0,0,0,0,0,0,0,0,0,4},
    {8,0,3,3,0,0,0,0,0,8,8,4,0,0,0,0,0,0,0,0,6},
    {8,0,0,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,6},
    {8,0,3,3,0,0,0,0,0,8,8,4,0,0,0,0,0,0,0,0,4},
    {8,0,0,0,0,0,0,0,0,8,4,0,0,0,0,0,6,6,6,0,6,4,6},
    {8,8,8,8,0,8,8,8,8,8,4,4,4,4,4,6,0,0,0,0,6},
    {7,7,7,7,0,7,7,7,0,8,0,8,0,8,0,8,4,0,4,0,6,0,6},
    {7,7,0,0,0,0,0,0,7,8,0,8,0,8,0,8,8,6,0,0,0,0,6},
    {7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,8,6,0,0,0,0,4},
    {7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,8,6,0,6,0,6,0,6},
    {7,7,0,0,0,0,0,0,7,8,0,8,0,8,0,8,8,6,4,6,0,6,6,6},
    {7,7,7,7,0,7,7,7,7,8,8,4,0,6,8,4,8,3,3,3,0,3,3,3},
    {2,2,2,2,0,2,2,2,2,4,6,4,0,0,6,0,6,3,0,0,0,0,0,3},
    {2,2,0,0,0,0,0,2,2,2,4,0,0,0,0,0,4,3,0,0,0,0,0,3},
    {2,0,0,0,0,0,0,2,4,0,0,0,0,0,0,4,3,0,0,0,0,0,3},
    {1,0,0,0,0,0,0,1,4,4,4,4,6,0,6,3,3,0,0,0,3,3},
    {2,0,0,0,0,0,0,2,2,2,1,2,2,2,2,6,6,0,0,5,0,5,0,5},
    {2,2,0,0,0,0,0,2,2,2,0,0,0,2,2,0,5,0,5,0,0,0,5,5},
    {2,0,0,0,0,0,0,2,0,0,0,0,0,2,5,0,5,0,5,0,5,0,5},
    {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,5},
    {2,0,0,0,0,0,0,2,0,0,0,0,0,2,5,0,5,0,5,0,5,0,5},
    {2,2,0,0,0,0,0,2,2,2,0,0,0,2,2,0,5,0,5,0,0,0,5,5},
    {2,2,2,2,1,2,2,2,2,2,1,2,2,2,2,5,5,5,5,5,5,5,5}
};
```

```

Jint32 buffer[screenHeight][screenWidth]; // y-coordinate first because it works per scanline

int main(int /*argc*/, char /*argv*/[])
{
    double posX = 22.0, posY = 11.5; //x and y start position
    double dirX = -1.0, dirY = 0.0; //initial direction vector
    double planeX = 0.0, planeY = 0.66; //the 2d raycaster version of camera plane

    double time = 0; //time of current frame
    double oldTime = 0; //time of previous frame

    std::vector<Uint32> texture[8];
    for(int i = 0; i < 8; i++) texture[i].resize(texWidth * texHeight);

    screen(screenWidth, screenHeight, 0, "Raycaster");

    //load some textures
    unsigned long tw, th, error = 0;
    error |= loadImage(texture[0], tw, th, "pics/eagle.png");
    error |= loadImage(texture[1], tw, th, "pics/redbrick.png");
    error |= loadImage(texture[2], tw, th, "pics/purplestone.png");
    error |= loadImage(texture[3], tw, th, "pics/greystone.png");
    error |= loadImage(texture[4], tw, th, "pics/bluestone.png");
    error |= loadImage(texture[5], tw, th, "pics/mossy.png");
    error |= loadImage(texture[6], tw, th, "pics/wood.png");
    error |= loadImage(texture[7], tw, th, "pics/colorstone.png");
    if(error) { std::cout << "error loading images" << std::endl; return 1; }

    //start the main loop
    while(!done())
    {

```

## b. Deuxième Partie :

Vient maintenant **le nouveau code de moulage au sol, ligne par ligne de bande verticale par bande verticale.**

La formule pour **rowDistance**, c'est-à-dire **la distance horizontale de la caméra au sol pour la ligne actuelle, qui est posZ / p** avec **p** la distance actuelle en pixels du centre de l'écran, peut être expliquée comme suit :

- **Le rayon de la caméra passe par les deux points suivantes : la caméra elle-même**, qui est à une certaine hauteur (**posZ**), et **un point devant la caméra** (à travers un plan vertical imaginaire contenant les pixels de l'écran) avec une distance horizontale 1 de la caméra et une position verticale **p** inférieure à **posZ** (**posZ - p**).
- **En passant par ce point, la ligne s'est déplacée verticalement par p unités et horizontalement par 1 unité** (pour toucher le sol, il doit à la place voyager par des unités **posZ** et parcourra le même rapport horizontalement).
- **Le rapport était de 1 / p pour traverser le plan de la caméra, donc afin d'atteindre le sol en allant plus loin de posZ, on obtient la distance horizontale totale de posZ / p.**

**Remarque :** Le pas à pas effectué ici est le mappage de texture affiné, ce qui signifie que l'on peut interpoler linéairement entre deux points plutôt que de devoir calculer une division différente pour chaque pixel.

```

//FLOOR CASTING
for(int y = 0; y < h; y++)
{
    // rayDir for leftmost ray (x = 0) and rightmost ray (x = w)
    float rayDirX0 = dirX - planeX;
    float rayDirY0 = dirY - planeY;
    float rayDirX1 = dirX + planeX;
    float rayDirY1 = dirY + planeY;

    // Current y position compared to the center of the screen (the horizon)
    int p = y - screenHeight / 2;

    // Vertical position of the camera.
    float posZ = 0.5 * screenHeight;

    // Horizontal distance from the camera to the floor for the current row.
    // 0.5 is the z position exactly in the middle between floor and ceiling.
    float rowDistance = posZ / p;

    // calculate the real world step vector we have to add for each x (parallel to camera p)
    // adding step by step avoids multiplications with a weight in the inner loop
    float floorStepX = rowDistance * (rayDirX1 - rayDirX0) / screenWidth;
    float floorStepY = rowDistance * (rayDirY1 - rayDirY0) / screenWidth;

    // real world coordinates of the leftmost column. This will be updated as we step to the right
    float floorX = posX + rowDistance * rayDirX0;
    float floorY = posY + rowDistance * rayDirY0;

    for(int x = 0; x < screenWidth; ++x)
    {
        // the cell coord is simply got from the integer parts of floorX and floorY
        int cellX = (int)(floorX);
        int cellY = (int)(floorY);

        // get the texture coordinate from the fractional part
        int tx = (int)(texWidth * (floorX - cellX)) & (texWidth - 1);
        int ty = (int)(texHeight * (floorY - cellY)) & (texHeight - 1);

        floorX += floorStepX;
        floorY += floorStepY;

        // choose texture and draw the pixel
        int floorTexture = 3;
        int ceilingTexture = 6;
        Uint32 color;

        // floor
        color = texture[floorTexture][texWidth * ty + tx];
        color = (color >> 1) & 8355711; // make a bit darker
        buffer[y][x] = color;

        //ceiling (symmetrical, at screenHeight - y - 1 instead of y)
        color = texture[ceilingTexture][texWidth * ty + tx];
        color = (color >> 1) & 8355711; // make a bit darker
        buffer[screenHeight - y - 1][x] = color;
    }
}

```

#### **4. Troisième étape :**

Vient ensuite **le code de moulage mural**, c'est exactement le même que précédemment (celui-ci va bande verticale par bande vertical, et pas ligne par ligne) :

```
//WALL CASTING
for(int x = 0; x < w; x++)
{
    //calculate ray position and direction
    double cameraX = 2 * x / double(w) - 1; //x-coordinate in camera space
    double rayDirX = dirX + planeX * cameraX;
    double rayDirY = dirY + planeY * cameraX;

    //which box of the map we're in
    int mapX = int(posX);
    int mapY = int(posY);

    //length of ray from current position to next x or y-side
    double sideDistX;
    double sideDistY;

    //length of ray from one x or y-side to next x or y-side
    double deltaDistX = std::abs(1 / rayDirX);
    double deltaDistY = std::abs(1 / rayDirY);
    double perpWallDist;

    //what direction to step in x or y-direction (either +1 or -1)
    int stepX;
    int stepY;

    int hit = 0; //was there a wall hit?
    int side; //was a NS or a EW wall hit?
```

```

//calculate step and initial sideDist
if (rayDirX < 0)
{
    stepX = -1;
    sideDistX = (posX - mapX) * deltaDistX;
}
else
{
    stepX = 1;
    sideDistX = (mapX + 1.0 - posX) * deltaDistX;
}
if (rayDirY < 0)
{
    stepY = -1;
    sideDistY = (posY - mapY) * deltaDistY;
}
else
{
    stepY = 1;
    sideDistY = (mapY + 1.0 - posY) * deltaDistY;
}
//perform DDA
while (hit == 0)
{
    //jump to next map square, OR in x-direction, OR in y-direction
    if (sideDistX < sideDistY)
    {
        sideDistX += deltaDistX;
        mapX += stepX;
        side = 0;
    }
    else
    {
        sideDistY += deltaDistY;
        mapY += stepY;
        side = 1;
    }
    //Check if ray has hit a wall
    if (worldMap[mapX][mapY] > 0) hit = 1;
}

//Calculate distance of perpendicular ray (Euclidean distance will give fisheye effect!)
if (side == 0) perpWallDist = (mapX - posX + (1 - stepX) / 2) / rayDirX;
else           perpWallDist = (mapY - posY + (1 - stepY) / 2) / rayDirY;

//Calculate height of line to draw on screen
int lineHeight = (int)(h / perpWallDist);

//calculate lowest and highest pixel to fill in current stripe
int drawStart = -lineHeight / 2 + h / 2;
if(drawStart < 0) drawStart = 0;
int drawEnd = lineHeight / 2 + h / 2;
if(drawEnd >= h) drawEnd = h - 1;
//texturing calculations
int texNum = worldMap[mapX][mapY] - 1; //1 subtracted from it so that texture 0 can be used!

//calculate value of wallX
double wallX; //where exactly the wall was hit
if (side == 0) wallX = posY + perpWallDist * rayDirY;
else           wallX = posX + perpWallDist * rayDirX;
wallX -= floor((wallX));

//x coordinate on the texture
int texX = int(wallX * double(texWidth));
if(side == 0 && rayDirX > 0) texX = texWidth - texX - 1;
if(side == 1 && rayDirY < 0) texX = texWidth - texX - 1;

// How much to increase the texture coordinate per screen pixel
double step = 1.0 * texHeight / lineHeight;
// Starting texture coordinate
double texPos = (drawStart - h / 2 + lineHeight / 2) * step;
for(int y = drawStart; y<drawEnd; y++)
{
    // Cast the texture coordinate to integer, and mask with (texHeight - 1) in case of overflow
    int texY = (int)texPos & (texHeight - 1);
    texPos += step;
    Uint32 color = texture[texNum][texWidth * texY + texX];
    //make color darker for y-sides: R, G and B byte each divided through two with a "shift" and an "and"
    if(side == 1) color = (color >> 1) & 8355711;
    buffer[y][x] = color;
}

```

---

## **5. Dernière Étape :**

Enfin, l'écran est dessiné et effacé à nouveau, et l'entrée est gérée :

```
drawBuffer(buffer[0]);
for(int y = 0; y < h; y++) for(int x = 0; x < w; x++) buffer[y][x] = 0; //clear the buffer instead of clearing the screen

//timing for input and FPS counter
oldTime = time;
time = getTicks();
double frameTime = (time - oldTime) / 1000.0; //frametime is the time this frame has taken, in seconds
print(1.0 / frameTime); //FPS counter
redraw();

//speed modifiers
double moveSpeed = frameTime * 3.0; //the constant value is in squares/second
double rotSpeed = frameTime * 2.0; //the constant value is in radians/second
readKeys();
//move forward if no wall in front of you
if (keyDown(SDLK_UP))
{
    if(worldMap[int(posX + dirX * moveSpeed)][int(posY)] == false) posX += dirX * moveSpeed;
    if(worldMap[int(posX)][int(posY + dirY * moveSpeed)] == false) posY += dirY * moveSpeed;
}
//move backwards if no wall behind you
if (keyDown(SDLK_DOWN))
{
    if(worldMap[int(posX - dirX * moveSpeed)][int(posY)] == false) posX -= dirX * moveSpeed;
    if(worldMap[int(posX)][int(posY - dirY * moveSpeed)] == false) posY -= dirY * moveSpeed;
}
//rotate to the right
if (keyDown(SDLK_RIGHT))
{
    //both camera direction and camera plane must be rotated
    double oldDirX = dirX;
    dirX = dirX * cos(-rotSpeed) - dirY * sin(-rotSpeed);
    dirY = oldDirX * sin(-rotSpeed) + dirY * cos(-rotSpeed);
    double oldPlaneX = planeX;
    planeX = planeX * cos(-rotSpeed) - planeY * sin(-rotSpeed);
    planeY = oldPlaneX * sin(-rotSpeed) + planeY * cos(-rotSpeed);
}
//rotate to the left
if (keyDown(SDLK_LEFT))
{
    //both camera direction and camera plane must be rotated
    double oldDirX = dirX;
    dirX = dirX * cos(rotSpeed) - dirY * sin(rotSpeed);
    dirY = oldDirX * sin(rotSpeed) + dirY * cos(rotSpeed);
    double oldPlaneX = planeX;
    planeX = planeX * cos(rotSpeed) - planeY * sin(rotSpeed);
    planeY = oldPlaneX * sin(rotSpeed) + planeY * cos(rotSpeed);
}
```

## 6. Résultat et Tricks :



Ces astuces ne sont en fait pas si spéciales, ce sont juste des choses qu'on peut modifier pour obtenir d'autres résultats :

- **Pour redimensionner les textures du sol et du plafond**, 4 fois par exemple, on peut modifier cette partie du code :

Ca ...

```
int floorTexX, floorTexY;  
floorTexX = int(currentFloorX * texWidth) % texWidth;  
floorTexY = int(currentFloorY * texHeight) % texHeight;
```

Par ca ...

```
int floorTexX, floorTexY;  
floorTexX = int(currentFloorX * texWidth / 4) % texWidth;  
floorTexY = int(currentFloorY * texHeight / 4) % texHeight;
```

- Jusqu'à présent, tout le niveau avait la même texture de sol partout. Puisque, de la manière dont le niveau est décrit, tous les non murs ont le code 0, cela ne peut pas être utilisé pour donner à chaque carré sa propre texture floortile. **On peut créer des carreaux non mureaux à 0 ou négatifs à la place, puis lors de la diffusion de rayons, un nombre négatif ne signifie pas de mur, et la valeur peut être utilisée pour indiquer quelle texture de sol doit être utilisée là-bas.** Si on souhaite faire de même avec le plafond, on aura également besoin d'une autre

valeur pour les textures du plafond. De fait, on peut envisager d'**utiliser une carte distincte pour les murs, le sol et le plafond**.

- Au lieu de faire on peut recourir à une autre méthode qui permettra de donner à chaque tuile sa propre texture en fonction de ses coordonnées : si la somme de ses coordonnées x et y sur la carte est pair, elle obtient la texture 3, si elle est impaire, elle obtient la texture 4, donnant un motif damier. Pour obtenir les coordonnées x et y de la tuile actuelle dans la carte, il faut prendre la partie entière de currentFloorX et currentFloorY. Pour cela, la boucle for de la pièce coulée au sol est modifiées en ceci (les parties en gras sont les nouvelles / modifiées) :

```
//draw the floor from drawEnd to the bottom of the screen
for(int y = drawEnd + 1; y < h; y++)
{
    currentDist = h / (2.0 * y - h); //you could make a small lookup table for this instead

    double weight = (currentDist - distPlayer) / (distWall - distPlayer);

    double currentFloorX = weight * floorXWall + (1.0 - weight) * posX;
    double currentFloorY = weight * floorYWall + (1.0 - weight) * posY;

    int floorTexX, floorTexY;
    floorTexX = int(currentFloorX * texWidth) % texWidth;
    floorTexY = int(currentFloorY * texHeight) % texHeight;

    int checkerBoardPattern = (int(currentFloorX) + int(currentFloorY)) % 2;
    int floorTexture;
    if(checkerBoardPattern == 0) floorTexture = 3;
    else floorTexture = 4;

    //floor
    buffer[y][x] = (texture[floorTexture][texWidth * floorTexY + floorTexX] >> 1) & 8355711;
    //ceiling (symmetrical)
    buffer[h - y][x] = texture[6][texWidth * floorTexY + floorTexX];
}
}
```

- De la même manière, il est également possible de **choisir la texture du sol pour chaque tuile en fonction d'une à la place** où la partie entière de currentFloorX donne les coordonnées du floortile courant dans la carte, tandis que la partie fractionnaire donne les coordonnées du texel sur la texture. Si on **modifie le code en damier de "(int (currentFloorX) + int (currentFloorY)) % 2"** en "**(int (currentFloorX + currentFloorY)) % 2**" : **on obtient pas un motif damier mais des rayures diagonales à la place**, car maintenant, les parties fractionnaire sont également ajoutées.
-

## **7. Code Version Verticale :**

---

### **a. Introduction :**

Comme **alternative à la technique de coulée au sol basée sur une ligne de balayage horizontale**, il est également possible de **travailler verticalement**. Cela permet de continuer à dessiner la même bande verticale qu'une bande murale actuelle a été dessinée.

Cependant, cette technique est **plus lente** car elle nécessite un mappage de texture correct en perspective. De fait, la technique basée sur la ligne de balayage est plus rapide à rendre grâce à la localité pour la mise en cache de la mémoire. **Le résultat est le même** (les sont sont toujours horizontaux), **il est juste rendu d'une manière différente**.

---

### **b. Comment cela marche ?**

Cette technique fonctionne comme suit :

- **Après avoir dessiné une bande verticale à partir du mur, on effectue le moulage du sol pour chaque pixel sur le pixel du mur inférieur jusqu'au bas de l'écran.**
- On doit connaître les coordonnées exactes de deux points du sol qui se trouvent à l'intérieur de la bande actuelle, les deux principaux points qui sont facilement trouvables : la position du joueur et le point du sol juste en face du mur.
- Ensuite, pour chaque pixel, il faut calculer la distance que sa projection sur le a par rapport au joueur. Avec cette distance, on peut trouver l'emplacement exact du sol que le pixel représente en utilisant une interpolation linéaire entre les deux points que l'on vient de trouver.
- Une fois que l'on a fait tous les calculs de sol, hors la position exacte, on peut facilement trouver les coordonnées du texel à partir de la texture pour obtenir la couleur du pixel que l'on doit dessiner. Comme le sol et le plafond sont symétriques, on sait que les coordonnées de texel de la texture du plafond sont les mêmes : on les dessine tout simplement au pixel correspondant dans la moitié supérieure de l'écran et on peut utiliser une texture différente pour le plafond et le sol.
- La distance entre la projection du pixel actuel et le sol peut être calculée comme suit : si le pixel est en bas de l'écran, on peut choisir une certaine distance (comme 1). Ainsi, tous les pixels entre ceux-ci sont donc compris entre 1 et l'infini. La distance que le pixel représente peut être définie en fonction de sa hauteur dans la moitié inférieure de l'écran est inversement tout en étant proportionnelle à 1 / hauteur. On peut utiliser la formule “**currentDist = h / 52.0 \* y - h**” pour calculer la distance du pixel actuel.
- On peut également précalculer une table de recherche pour cela à la place, car il n'y a que h / 2 valeurs possibles (une moitié de l'écran dans le sens vertical).
- L'interpolation linéaire, pour obtenir l'emplacement exact du sol en fonction de la distance actuelle et des deux distances connues, peut être effectuée avec un facteur de pondération. Ce facteur de pondération est “**weight = (currentDist - disPlayer) / (distWall - distPlayer)**”, et comme le pixel actuel sera toujours entre le mur et la position du lecteur, la position exacte est alors : “**currentFloorPos =**

wieght \* floorPosWall + (1.0 - poids) \* playerPos". Noter que distPlayer vaut en fait 0, donc le poids est en fait "currentDist/distWall".

- La coulée au sol se fait donc maintenant juste après la coulée du mur, dans la même boucle en x.
- Tout d'abord, la position du sol juste ne face du mur est calculée (sachant qu'il y a 4 cas différents possibles selon si un côté est nord, est, sud ou ouest d'un mur à été touché).
- Une fois cette position et les distances définies, la boucle for dans la direction y qui va du pixel sous le mur jusqu'au bas de l'écran commence : il calcule la distance actuelle, en dehors du poids, à partir de la position exacte du sol, et en dehors de cela la coordonnée du texel sur la texture.
- Avec cette information, un pixel de plancher et un pixel de plafond peuvent être dessinés (le sol est assombri).

```

for(int x = 0; x < w; x++)
{
    //WALL CASTING
    // [SNIP...] the floor casting code goes in the same x-for-loop as the wall casting, wall casting code not duplicated here

    //FLOOR CASTING (vertical version, directly after drawing the vertical wall stripe for the current x)
    double floorXWall, floorYWall; //x, y position of the floor texel at the bottom of the wall

    //4 different wall directions possible
    if(side == 0 && rayDirX > 0)
    {
        floorXWall = mapX;
        floorYWall = mapY + wallX;
    }
    else if(side == 0 && rayDirX < 0)
    {
        floorXWall = mapX + 1.0;
        floorYWall = mapY + wallX;
    }
    else if(side == 1 && rayDirY > 0)
    {
        floorXWall = mapX + wallX;
        floorYWall = mapY;
    }
    else
    {
        floorXWall = mapX + wallX;
        floorYWall = mapY + 1.0;
    }

    double distWall, distPlayer, currentDist;

    distWall = perpWallDist;
    distPlayer = 0.0;

    if (drawEnd < 0) drawEnd = h; //becomes < 0 when the integer overflows

    //draw the floor from drawEnd to the bottom of the screen
    for(int y = drawEnd + 1; y < h; y++)
    {
        currentDist = h / (2.0 * y - h); //you could make a small lookup table for this instead

        double weight = (currentDist - distPlayer) / (distWall - distPlayer);

        double currentFloorX = weight * floorXWall + (1.0 - weight) * posX;
        double currentFloorY = weight * floorYWall + (1.0 - weight) * posY;

        int floorTexX, floorTexY;
        floorTexX = int(currentFloorX * texWidth) % texWidth;
        floorTexY = int(currentFloorY * texHeight) % texHeight;

        //floor
        buffer[y][x] = (texture[3][texWidth * floorTexY + floorTexX] >> 1) & 8355711;
        //ceiling (symmetrical)
        buffer[h - y][x] = texture[6][texWidth * floorTexY + floorTexX];
    }
}

```

## VII. Etape 5 : Raycasting III (Sprites)

---

### 1. Introduction :

Précédemment, on a vu comment créer des murs et des sols texturés. Cependant, dans un jeu, un monde seulement avec des murs et des sols reste un monde vide : pour qu'un jeu fonctionne, il doit y avoir **des goodies, des ennemis, des objets** comme des tonneaux ou des arbres.

Ceux-ci ne peuvent pas être dessinés comme un mur ou un sol mais ils ne peuvent pas être non plus dessinés en tant que modèles 3D. Au lieu de cela, on utilise **des sprites, des images 2D toujours face à au joueur qui deviennent plus petites si elles sont éloignées**. Elles sont donc faciles à dessiner et nécessitent une seule image.

---

### 2. Comment ça marche ?

La technique utilisée pour dessiner les sprites est totalement différente de la technique de raycasting. Au lieu, cela fonctionne de manière très similaire à la façon dont les sprites sont dessinés dans un moteur 2D avec des projections. De fait, on doit faire **une projection en 2D suivie de quelques techniques supplémentaires pour la combiner avec le raycasting**.

Le dessin des sprites est **effectué une fois que les murs et le sol sont déjà dessinés**. Les démarches à suivre sont les suivantes :

- **Lors de la diffusion de rayons sur les murs, il faut enregistrer la distance perpendiculaire de chaque bande verticale dans un ZBuffer 1D.**
- **Calculer la distance de chaque sprite au joueur.**
- **Utiliser cette distance pour trier les sprites**, du plus éloigné au plus proche de la caméra.
- **Projeter le sprite sur le plan de la caméra (2D) en soustrayant la position du joueur de la position du sprite, puis multiplier le résultat par l'inverse de la matrice de la caméra 2x2.**
- **Calculer la taille du sprite à l'écran** (à la fois dans la direction x et y) en utilisant la distance perpendiculaire.
- **Dessiner la bande verticale des sprites par bande verticale** (il ne faut pas dessiner la verticale si la distance est plus éloignée que ZBuffer 1D des murs de la bande actuelle).
- **Dessiner la bande verticale pixel par pixel** et s'assurer qu'il y a une couleur visible ou tous les seraient des rectangles (on n'est pas obligé de mettre à jour le ZBuffer pendant que l'on dessine les bandes, puisqu'elles sont triées : celles qui sont les plus proches du joueur seront dessinées en dernier, elles sont donc dessinées sur les plus éloignées).

Comme expliqué, ici, les sprites sont projetés à l'écran en 2D. Pour **obtenir les coordonnées du sprite dans l'espace de la caméra**, il faut **soustraire d'abord la position du joueur de la position du sprite pour obtenir la position du sprite par rapport au joueur**.

Ensuite, il **doit être tourné pour que la direction soit relative au joueur**. La caméra peut également être biaisée et avoir une certaine taille : ce n'est donc pas vraiment une rotation mais une **transformation**. La transformation se fait **en multipliant la position relative du sprite par l'inverse de la matrice de la caméra**.

La matrice de la caméra est ici :

```
[planeX    dirX]  
[planeY    dirY]
```

And the inverse of a 2x2 matrix is very easy to calculate

$$\frac{1}{(planeX*dirY - dirX*planeY)} \begin{bmatrix} dirY & -dirX \\ -planeY & planeX \end{bmatrix}$$

Ensuite, on **obtient les coordonnées X et Y du sprite dans l'espace caméra, où Y est la profondeur à l'intérieur de l'écran**. Pour projeter à l'écran, il faut diviser X par la profondeur, puis le traduire et le mettre à l'échelle.

Pour placer les objets dans le niveau, de nombreuses peuvent être faites :

- **Chaque objet peut avoir ses propres coordonnées en virgule flottante** (il n'est pas nécessaire qu'il soit exactement au centre des carreaux de sol).
- On peut faire **une liste de chaque objet et donner ses coordonnées et sa texture un par un**.
- On peut également **placer les objets en créant une deuxième carte** (un tableau 2D), **et pour chaque coordonnée de tuile, placer un ou aucun objet**, de la même manière que de placer des murs. Si l'on fait cela, il faut laisser le programme lire cette carte tout en créant une liste d'objets, chaque objet étant placé au centre de la tuile de la carte correspondante. Les coordonnées au centre d'une tuile sont divisées en deux, tandis que les coordonnées entières seront les coins des carreaux.

### 3. Le Code :

Le code reste similaire au code vu précédemment avec quelques modifications.

**Des nouvelles variables pour le casting de sprites sont stockés dans struct Sprite et qui contient la position texture du sprite :**

- La valeur **numSprites** représente le nombre de sprites.
- Le tableau de sprite définit les positions et textures de tous sprites (chaque troisième nombre est le numéro de sa texture).
- La fonction **bubbleSort** va trier sprites qui prend en argument tableaux **spriteOrder** et **spriteDistance**.
- ZBuffer est l'équivalent 1D du ZBuffer dans un vrai moteur

```
#define screenWidth 640
#define screenHeight 480
#define texWidth 64
#define texHeight 64
#define mapWidth 24
#define mapHeight 24

int worldMap[mapWidth][mapHeight] =
{
    {8,8,8,8,8,8,8,8,8,8,4,4,6,4,6,4,4,4,4,6,4},
    {8,0,0,0,0,0,0,0,0,0,8,4,0,0,0,0,0,0,0,0,0,4},
    {8,0,3,3,0,0,0,0,0,0,8,4,0,0,0,0,0,0,0,0,0,6},
    {8,0,0,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,6},
    {8,0,3,3,0,0,0,0,0,0,8,8,4,0,0,0,0,0,0,0,0,4},
    {8,0,0,3,3,0,0,0,0,0,8,8,4,0,0,0,0,0,0,0,0,4},
    {8,0,0,0,0,0,0,0,0,0,8,4,0,0,0,0,0,0,0,0,6,6,6,6,4,6},
    {8,8,8,8,8,8,8,8,8,8,4,4,4,4,4,4,6,0,0,0,0,6},
    {7,7,7,7,0,7,7,7,7,0,8,0,8,0,8,4,0,4,0,6,0,6},
    {7,7,0,0,0,0,0,0,0,7,8,0,8,0,8,8,6,0,0,0,0,6},
    {7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,8,6,0,0,0,0,4},
    {7,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,8,6,0,6,0,6,6},
    {7,7,0,0,0,0,0,0,7,8,0,8,0,8,8,8,6,4,6,0,6,6,6},
    {7,7,7,7,0,7,7,7,7,8,8,4,0,6,8,4,8,3,3,0,3,3,3},
    {2,2,2,2,0,2,2,2,2,4,6,4,0,0,6,0,6,3,0,0,0,0,3},
    {2,2,0,0,0,0,0,2,2,4,0,0,0,0,0,0,4,3,0,0,0,0,3},
    {2,0,0,0,0,0,0,2,4,0,0,0,0,0,0,4,3,0,0,0,0,3},
    {1,0,0,0,0,0,0,0,1,4,4,4,4,4,6,0,6,3,3,0,0,0,3,3},
    {2,0,0,0,0,0,0,2,2,2,1,2,2,2,6,6,0,0,5,0,5,0,5},
    {2,2,0,0,0,0,2,2,2,0,0,0,2,2,0,5,0,5,0,0,0,5,5},
    {2,0,0,0,0,0,0,2,0,0,0,0,0,0,2,5,0,5,0,5,0,5,0,5},
    {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,5},
    {2,0,0,0,0,0,0,2,0,0,0,0,0,0,2,5,0,5,0,5,0,5,0,5},
    {2,2,0,0,0,0,0,2,2,2,0,0,0,2,2,0,5,0,5,0,0,0,5,5},
    {2,2,2,2,1,2,2,2,2,2,1,2,2,2,5,5,5,5,5,5,5,5,5}
};

struct Sprite
{
    double x;
    double y;
    int texture;
};
```

I a  
et la  
les  
des  
3D.

```
#define numSprites 19
Sprite sprite[numSprites] =
{
    {20.5, 11.5, 10}, //green light in front of playerstart
    //green lights in every room
    {18.5, 4.5, 10},
    {10.0, 4.5, 10},
    {10.0, 12.5, 10},
    {3.5, 6.5, 10},
    {3.5, 20.5, 10},
    {3.5, 14.5, 10},
    {14.5, 20.5, 10},
    //row of pillars in front of wall: fisheye test
    {18.5, 10.5, 9},
    {18.5, 11.5, 9},
    {18.5, 12.5, 9},
    //some barrels around the map
    {21.5, 1.5, 8},
    {15.5, 1.5, 8},
    {16.0, 1.8, 8},
    {16.2, 1.2, 8},
    {3.5, 2.5, 8},
    {9.5, 15.5, 8},
    {10.0, 15.1, 8},
    {10.5, 15.8, 8},
},
Jint32 buffer[screenHeight][screenWidth]; // y-coordinate first because it works per scanline
//1D Zbuffer
double ZBuffer[screenWidth];
//arrays used to sort the sprites
int spriteOrder[numSprites];
double spriteDistance[numSprites];
//function used to sort the sprites
void sortSprites(int* order, double* dist, int amount);
int main(int argc, char *argv[])
{
    double posX = 22.0, posy = 11.5; //x and y start position
    double dirX = -1.0, dirY = 0.0; //initial direction vector
    double planeX = 0.0, planeY = 0.66; //the 2d raycaster version of camera plane

    double time = 0; //time of current frame
    double oldTime = 0; //time of previous frame

    std::vector<Uint32> texture[11];
    for(int i = 0; i < 11; i++) texture[i].resize(texWidth * texHeight);
```

#### a. Rajoute des textures :

3 nouvelles textures sont chargées pour les sprites :

```

screen(screenWidth, screenHeight, 0, "Raycaster");

//load some textures
unsigned long tw, th, error = 0;
error |= loadImage(texture[0], tw, th, "pics/eagle.png");
error |= loadImage(texture[1], tw, th, "pics/redbrick.png");
error |= loadImage(texture[2], tw, th, "pics/purplestone.png");
error |= loadImage(texture[3], tw, th, "pics/greystone.png");
error |= loadImage(texture[4], tw, th, "pics/bluestone.png");
error |= loadImage(texture[5], tw, th, "pics/mossy.png");
error |= loadImage(texture[6], tw, th, "pics/wood.png");
error |= loadImage(texture[7], tw, th, "pics/colorstone.png");

//load some sprite textures
error |= loadImage(texture[8], tw, th, "pics/barrel.png");
error |= loadImage(texture[9], tw, th, "pics/pillar.png");
error |= loadImage(texture[10], tw, th, "pics/greenlight.png");
if(error) { std::cout << "error loading images" << std::endl; return 1; }

```

## b. Boucle Principale :

**La boucle principale**, qui commence par le raycasting des sols et des murs (même code que précédemment) :

```

//start the main loop
while(!done())
{
    //FLOOR CASTING
    for(int y = 0; y < h; y++)
    {
        // rayir for leftmost ray (x = 0) and rightmost ray (x = w)
        float rayirX0 = dirx - planex;
        float rayirY0 = diry - planeY;
        float rayirX1 = dirx + planex;
        float rayirY1 = diry + planeY;

        // Current y position compared to the center of the screen (the horizon)
        int p = y - screenHeight / 2;

        // Vertical position of the camera.
        float pos1 = 0.5 * screenHeight;

        // Horizontal distance from the camera to the floor for the current row.
        // 0.5 z position exactly in the middle between floor and ceiling.
        float rowdistance = pos1 / p;

        // calculate the real world step vector we have to add for each x (parallel to camera plane)
        // adding step by step avoids multiplications with a weight in the inner loop
        float floorStepx = rowdistance * (rayirX1 - rayirX0) / screenWidth;
        float floorStepy = rowdistance * (rayirY1 - rayirY0) / screenWidth;

        // real world coordinates of the leftmost column. This will be updated as we step to the right.
        float floorX = pos1 + rowdistance * rayirX0;
        float floorY = pos1 + rowdistance * rayirY0;

        for(int x = 0; x < screenWidth; ++x)
        {
            // the cell coord is simply got from the integer parts of floorX and floorY
            int cellX = (int)(floorX);
            int cellY = (int)(floorY);

            // get the texture coordinate from the fractional part
            int tx = (int)(texwidth * (floorX - cellX)) & (texwidth - 1);
            int ty = (int)(texheight * (floorY - cellY)) & (texheight - 1);

            floorX += floorStepx;
            floorY += floorStepy;

            // choose texture and draw the pixel
            int floorTexture = 3;
            int ceilingTexture = 6;
            Uint32 color;

            // floor
            color = texture[floorTexture][texwidth * ty + tx];
            color = (color >> 1) & 8355711; // make a bit darker
            buffer1[x] = color;

            //ceiling (symmetrical, at screenHeight - y - 1 instead of y)
            color = texture[ceilingTexture][texwidth * ty + tx];
            color = (color >> 1) & 8355711; // make a bit darker
            buffer1[screenHeight - y - 1][x] = color;
        }
    }

    // WALL CASTING
    for(int x = 0; x < w; x++)
    {
        //calculate ray position and direction
        double camerax = 2 * x / double(w) - 1; //x-coordinate in camera space
        double rayirX = dirx + planex * camerax;
        double rayirY = diry + planeY * camerax;

        //which box of the map we're in
        int mapX = int(posX);
        int mapY = int(posY);

        //length of ray from current position to next x or y-side
        double sideDistX;
        double sideDistY;

        //length of ray from one x or y-side to next x or y-side
        double deltaDistX = std::abs(1 / rayirX);
        double deltaDistY = std::abs(1 / rayirY);
        double perpWallDist;

        //what direction to step in x or y-direction (either +1 or -1)
        int stepX;
        int stepY;

        int hit = 0; //was there a wall hit?
        int side; //was a NS or a EW wall hit?

        //calculate step and initial sideDist
        if(rayirX < 0)
        {
            stepX = -1;
            sideDistX = (posX - mapX) * deltaDistX;
        }
        else
        {
            stepX = 1;
            sideDistX = (mapX + 1.0 - posX) * deltaDistX;
        }
        if(rayirY < 0)
        {
            stepY = -1;
            sideDistY = (posY - mapY) * deltaDistY;
        }
        else
        {
            stepY = 1;
            sideDistY = (mapY + 1.0 - posY) * deltaDistY;
        }

        //perform DDA
        while(hit == 0)
        {
            //jump to next map square, OR in x-direction, OR in y-direction
            if(sideDistX < sideDistY)
            {
                sideDistX += deltaDistX;
                mapX += stepX;
                side = 0;
            }
            else
            {
                sideDistY += deltaDistY;
                mapY += stepY;
                side = 1;
            }

            //check if ray has hit a wall
            if(worldMap[mapX][mapY] > 0) hit = 1;
        }
    }
}

```

```

//Calculate distance of perpendicular ray (Euclidean distance will give fisheye effect!)
if (side == 0) perpWallDist = (mapX - posX + (1 - stepX) / 2) / rayDirX;
else           perpWallDist = (mapY - posY + (1 - stepY) / 2) / rayDirY;

//Calculate height of line to draw on screen
int lineHeight = (int)(h / perpWallDist);

//calculate lowest and highest pixel to fill in current stripe
int drawStart = -lineHeight / 2 + h / 2;
if(drawStart < 0) drawStart = 0;
int drawEnd = lineHeight / 2 + h / 2;
if(drawEnd >= h) drawEnd = h - 1;
//texturing calculations
int texNum = worldMap[mapX][mapY] - 1; //1 subtracted from it so that texture 0 can be used!

//calculate value of wallX
double wallX; //where exactly the wall was hit
if (side == 0) wallX = posY + perpWallDist * rayDirY;
else           wallX = posX + perpWallDist * rayDirX;
wallX -= floor((wallX));

//x coordinate on the texture
int texX = int(wallX * double(texWidth));
if(side == 0 && rayDirX > 0) texX = texWidth - texX - 1;
if(side == 1 && rayDirY < 0) texX = texWidth - texX - 1;

// How much to increase the texture coordinate per screen pixel
double step = 1.0 * texHeight / lineHeight;
// Starting texture coordinate
double texPos = (drawStart - h / 2 + lineHeight / 2) * step;
for(int y = drawStart; y<drawEnd; y++)
{
    // Cast the texture coordinate to integer, and mask with (texHeight - 1) in case of overflow
    int texY = (int)texPos & (texHeight - 1);
    texPos += step;
    int color = texture[texNum][texWidth * texY + texX];
    //make color darker for y-sides: R, G and B byte each divided through two with a 'shift' and an 'and'
    if(side == 1) color = (color >> 1) & 8355711;
    buffer[y][x] = color;
}

```

### c. ZBuffer :

Après avoir rayé le mur, le **ZBuffer doit être réglé**. Ce ZBuffer est **1D** car il ne contient que la distance au mur de chaque bande verticale, au lieu de l'avoir pour chaque pixel. Cela termine également la boucle à travers chaque bande verticale, car le rendu des sprites se fera en dehors de cette boucle.

```

//SET THE ZBUFFER FOR THE SPRITE CASTING
ZBuffer[x] = perpWallDist; //perpendicular distance is used
}

```

#### d. Les Sprites :

Une fois les sols et les murs dessinés, les sprites peuvent être dessinés :

- Il faut **trier les sprites de loin à près**, de sorte que les plus éloignés soient dessinés en premier (la distance calculée pour le tri des sprites n'est jamais utilisée par la suite, car la distance perpendiculaire est utilisée à la place).
- **Calculer la taille que les sprites devraient avoir à l'écran puis les dessiner bande par bande** (la multiplication matricielle pour la projection est très facile car il ne s'agit que d'une matrice 2x2).

```

//SPRITE CASTING
//sort sprites from far to close
for(int i = 0; i < numSprites; i++)
{
    spriteOrder[i] = i;
    spriteDistance[i] = ((posX - sprite[i].x) * (posX - sprite[i].x) + (posY - sprite[i].y) * (posY - sprite[i].y)); //sqrt not needed
}
sortSprites(spriteOrder, spriteDistance, numSprites);

//after sorting the sprites, do the projection and draw them
for(int i = 0; i < numSprites; i++)
{
    //translate sprite position to relative to camera
    double spriteX = sprite[spriteOrder[i]].x - posX;
    double spriteY = sprite[spriteOrder[i]].y - posY;

    //transform sprite with the inverse camera matrix
    // [ planeX   dirX ] -1      [ dirY       -dirX ]
    // [           ] = 1/(planeX*dirY-dirX*planeY) * [           ]
    // [ planeY   dirY ]           [ -planeY   planeX ]

    double invDet = 1.0 / (planeX * dirY - dirX * planeY); //required for correct matrix multiplication

    double transformX = invDet * (dirY * spriteX - dirX * spriteY);
    double transformY = invDet * (-planeY * spriteX + planeX * spriteY); //this is actually the depth inside the screen, that's why we invert it

    int spriteScreenX = int((w / 2) * (1 + transformX / transformY));

    //calculate height of the sprite on screen
    int spriteHeight = abs(int(h / (transformY))); //using 'transformY' instead of the real distance prevents fisheye
    //calculate lowest and highest pixel to fill in current stripe
    int drawStartY = -spriteHeight / 2 + h / 2;
    if(drawStartY < 0) drawStartY = 0;
    int drawEndY = spriteHeight / 2 + h / 2;
    if(drawEndY >= h) drawEndY = h - 1;

    //calculate width of the sprite
    int spriteWidth = abs(int(h / (transformY)));
    int drawStartX = -spriteWidth / 2 + spriteScreenX;
    if(drawStartX < 0) drawStartX = 0;
    int drawEndX = spriteWidth / 2 + spriteScreenX;
    if(drawEndX >= w) drawEndX = w - 1;

    //loop through every vertical stripe of the sprite on screen
    for(int stripe = drawStartX; stripe < drawEndX; stripe++)
    {
        int texX = int(256 * (stripe - (-spriteWidth / 2 + spriteScreenX)) * texWidth / spriteWidth) / 256;
        //the conditions in the if are:
        //1) it's in front of camera plane so you don't see things behind you
        //2) it's on the screen (left)
        //3) it's on the screen (right)
        //4) ZBuffer, with perpendicular distance
        if(transformY > 0 && stripe > 0 && stripe < w && transformY < zBuffer[stripe])
            for(int y = drawStartY; y < drawEndY; y++) //for every pixel of the current stripe
            {
                int d = (y) * 256 - h * 128 + spriteHeight * 128; //256 and 128 factors to avoid floats
                int texY = ((d * texHeight) / spriteHeight) / 256;
                Uint32 color = texture[sprite[spriteOrder[i]].texture][texWidth * texY + texX]; //get current color from the texture
                if((color & 0x00FFFFFF) != 0) buffer[y][stripe] = color; //paint pixel if it isn't black, black is the invisible color
            }
    }
}

```

```
//sort algorithm
//sort the sprites based on distance
void sortSprites(int* order, double* dist, int amount)
{
    std::vector<std::pair<double, int>> sprites(amount);
    for(int i = 0; i < amount; i++) {
        sprites[i].first = dist[i];
        sprites[i].second = order[i];
    }
    std::sort(sprites.begin(), sprites.end());
    // restore in reverse order to go from farthest to nearest
    for(int i = 0; i < amount; i++) {
        dist[i] = sprites[amount - i - 1].first;
        order[i] = sprites[amount - i - 1].second;
    }
}
```

---

## e. Mis à jour de l'écran et Saisie :

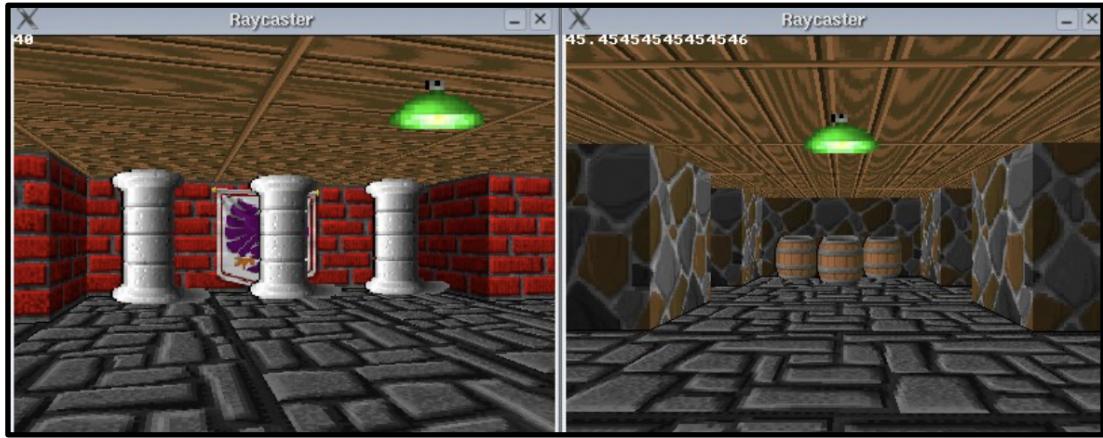
Une fois que tout est dessiné, l'écran est mis à jour et les touches de saisie sont gérées :

```
drawBuffer(buffer[0]);
for(int y = 0; y < h; y++) for(int x = 0; x < w; x++) buffer[y][x] = 0; //clear the buffer instead of cls()

//timing for input and FPS counter
oldTime = time;
time = getTicks();
double frameTime = (time - oldTime) / 1000.0; //frametime is the time this frame has taken, in seconds
print(1.0 / frameTime); //FPS counter
redraw();

//speed modifiers
double moveSpeed = frameTime * 3.0; //the constant value is in squares/second
double rotSpeed = frameTime * 2.0; //the constant value is in radians/second
readKeys();
//move forward if no wall in front of you
if (keyDown(SDLK_UP))
{
    if(worldMap[int(posX + dirX * moveSpeed)][int(posY)] == false) posX += dirX * moveSpeed;
    if(worldMap[int(posX)][int(posY + dirY * moveSpeed)] == false) posY += dirY * moveSpeed;
}
//move backwards if no wall behind you
if (keyDown(SDLK_DOWN))
{
    if(worldMap[int(posX - dirX * moveSpeed)][int(posY)] == false) posX -= dirX * moveSpeed;
    if(worldMap[int(posX)][int(posY - dirY * moveSpeed)] == false) posY -= dirY * moveSpeed;
}
//rotate to the right
if (keyDown(SDLK_RIGHT))
{
    //both camera direction and camera plane must be rotated
    double oldDirX = dirX;
    dirX = dirX * cos(-rotSpeed) - dirY * sin(-rotSpeed);
    dirY = oldDirX * sin(-rotSpeed) + dirY * cos(-rotSpeed);
    double oldPlaneX = planeX;
    planeX = planeX * cos(-rotSpeed) - planeY * sin(-rotSpeed);
    planeY = oldPlaneX * sin(-rotSpeed) + planeY * cos(-rotSpeed);
}
//rotate to the left
if (keyDown(SDLK_LEFT))
{
    //both camera direction and camera plane must be rotated
    double oldDirX = dirX;
    dirX = dirX * cos(rotSpeed) - dirY * sin(rotSpeed);
    dirY = oldDirX * sin(rotSpeed) + dirY * cos(rotSpeed);
    double oldPlaneX = planeX;
    planeX = planeX * cos(rotSpeed) - planeY * sin(rotSpeed);
    planeY = oldPlaneX * sin(rotSpeed) + planeY * cos(rotSpeed);
}
```

f. Résultat et Remarques :



La lumière verte est un très petit sprite, mais le programme passe toujours par tous ses pixels invisibles pour vérifier leur couleur. Cela pourrait être rendu plus rapide en indiquant quels sprites ont de grandes parties visibles et en ne dessinant qu'une partie rectangulaire plus petite contenant tous les pixels visibles.

Pour que certains objets ne puissent pas être dérangés, on peut soit vérifier la distance du joueur à chaque objet lorsqu'il se déplace pour la détection de collision, soit créer une autre carte 2D qui contient pour chaque carré s'il est débrouachable ou non, cela peut être utilisé pour les murs aussi.

Dans Wolfenstein 3D par exemple, certains objets (par exemple les soldats) ont 8 images différentes lorsqu'ils sont visualisés sous différents angles, pour donner l'impression que le sprite est vraiment 3D. On peut obtenir l'angle de l'objet par rapport au joueur par exemple avec la fonction atan2, puis choisir 1 des 8 textures en fonction de l'angle (on peut donner encore plus de textures au sprites pour l'animation).

---

a. Mise à l'échelle des sprites :

Il est assez facile de laisser le programme dessiner les sprites plus ou moins grands et de déplacer les sprites vers le haut ou vers le bas :

- Pour réduire le sprite : diviser spriteWidth et spriteHeight par quelque chose.
- Si on divise par deux la hauteur des sprites, un pilier par exemple, la bas se déplacera vers le haut pour que le pilier semble flotter.
- Outre les paramètres uDiv et vDic pour réduire le sprite, un paramètre vMove est également ajouté pour déplacer le sprite vers le bas s'il doit se tenir au sol, ou vers le haut s'il doit être suspendu au plafond (vMoveScreen est vMove projeté sur l'écran en le divisant par la profondeur).

```

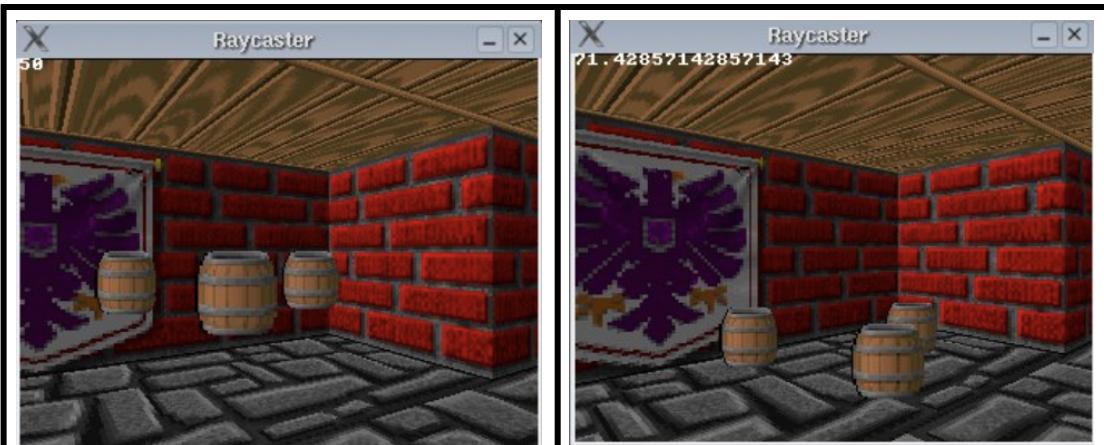
//parameters for scaling and moving the sprites
#define uDiv 1
#define vDiv 1
#define vMove 0.0
int vMoveScreen = int(vMove / transformY);

//calculate height of the sprite on screen
int spriteHeight = abs(int(h / (transformY))) / vDiv; //using 'transformY' instead of the real distance prevents fisheye
//calculate lowest and highest pixel to fill in current stripe
int drawStartY = -spriteHeight / 2 + h / 2 + vMoveScreen;
if(drawStartY < 0) drawStartY = 0;
int drawEndY = spriteHeight / 2 + h / 2 + vMoveScreen;
if(drawEndY >= h) drawEndY = h - 1;

//calculate width of the sprite
int spriteWidth = abs( int (h / (transformY)) ) / uDiv;
int drawStartX = -spriteWidth / 2 + spriteScreenX;
if(drawStartX < 0) drawStartX = 0;
int drawEndX = spriteWidth / 2 + spriteScreenX;
if(drawEndX >= w) drawEndX = w - 1;

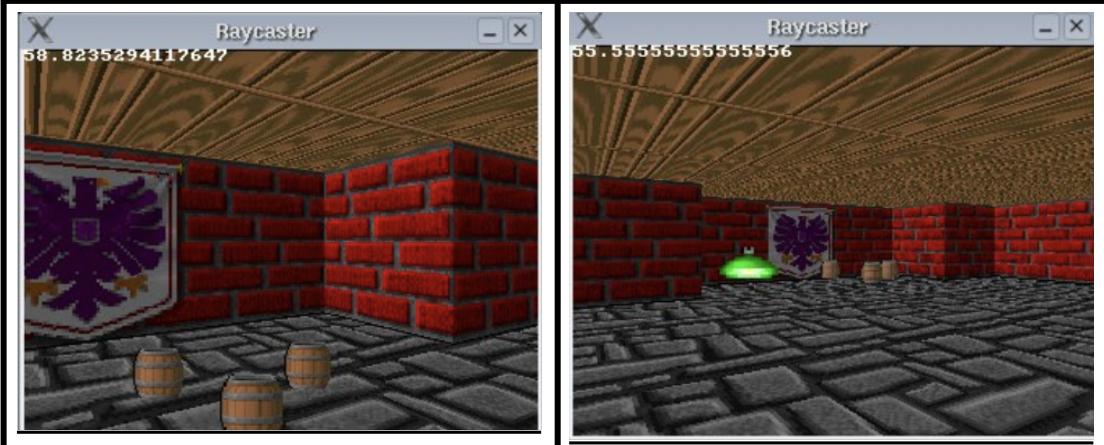
//loop through every vertical stripe of the sprite on screen
for(int stripe = drawStartX; stripe < drawEndX; stripe++)
{
    int texX = int(256 * (stripe - (-spriteWidth / 2 + spriteScreenX)) * texWidth / spriteWidth) / 256;
    //the conditions in the if are:
    //1) it's in front of camera plane so you don't see things behind you
    //2) it's on the screen (left)
    //3) it's on the screen (right)
    //4) ZBuffer, with perpendicular distance
    if(transformY > 0 && stripe > 0 && stripe < w && transformY < ZBuffer[stripe])
        for(int y = drawStartY; y < drawEndY; y++) //for every pixel of the current stripe
    {
        int d = (y-vMoveScreen) * 256 - h * 128 + spriteHeight * 128; //256 and 128 factors to avoid floats
        int texY = ((d * texHeight) / spriteHeight) / 256;
        Uint32 color = texture[sprite[spriteOrder[i].texture][texWidth * texY + texX]]; //get current color from the texture
        if((color & 0x00FFFFFF) != 0) buffer[y][stripe] = color; //paint pixel if it isn't black, black is the invisible color
    }
}
}

```



Lorsque  $uDiv = 2$ ,  $vDiv = 2$  et  $vMove = 0.0$ , les sprites sont deux fois moins grands et flottants.

On remet les sprites au sol en définissant  $vMove$  sur 64.0 (la taille de la texture).



Si on rend vMove encore plus haut pour placer les sprites sous le sol, ils seront toujours dessinés à travers le sol, car le ZBuffer est 1D et ne peut détecter qui si le sprite est devant ou derrière un mur.

Bien sûr, en abaissant les barils, le feu vert est également plus bas pour qu'il ne pende plus au plafond. Pour rendre cela utile, on doit donner à chaque sprite ses propres paramètres uDiv, vDuv et vMove, par exemple on peut les mettre dans la structure du sprite.

#### 4. Sprites Translucides :

Parce qu'on travaille en couleur RVB, **rendre les sprites translucides est très simple**. Tout ce que l'on doit faire est de **prendre la moyenne de l'ancienne couleur dans le tampon et de la nouvelle couleur du sprite**.

Les vieux jeux comme Wolfenstein 3D utilisent une palette de 256 couleurs sans règles mathématiques logiques pour les couleurs de la palette, donc la translucidité n'était pas si facile.

Pour cela, il faut changer ça...

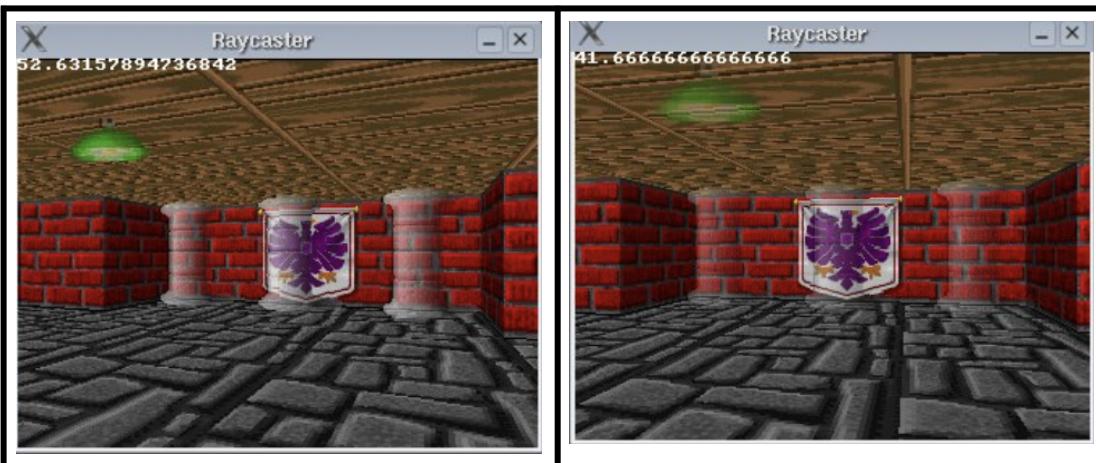
```
if((color & 0x00FFFFFF) != 0) buffer[y][stripe] = color; //paint pixel if it isn't black, black is the invisible color
```

En ca...

```
if((color & 0x00FFFFFF) != 0) buffer[y][stripe] = RGBtoINT(INTtoRGB(buffer[y][stripe]) / 2 + INTtoRGB(color) / 2);
```

Pour rendre encore plus transparent...

```
if((color & 0x00FFFFFF) != 0) buffer[y][stripe] = RGBtoINT(3 * INTtoRGB(buffer[y][stripe]) / 4 + INTtoRGB(color) / 4);
```



Enfin, on peut également essayer des astuces plus spéciales, par exemple des sprites translucides qui donnent aux murs derrière eux une couleur négative (de manière plus

utiles, il serait plus pratique donner à chaque sprite son propre effet de translucidité avec un paramètre supplémentaire dans la structure du sprite) :

```
if((color & 0x00FFFFFF) != 0) buffer[y][stripe] = RGBtoINT((RGB_White - INTtoRGB(buffer[y][stripe])) / 2 + INTtoRGB(color) / 2);
```



## VIII. Récapitulatif :

### 1. Le Raycasting :

#### a. Théorie :

En théorie :

- **J'envoie des rayons de gauche à droite depuis la position du joueur** ( sachant qu'au lieu de lancer un rayon pour chaque pixel, on lance un rayon par colonnes);
- **Plus le rayonnement du temps à atteindre le mur, plus il est loin;**
- **Plus la distance au mur est grande, plus la colonne de pixels est petite.**

#### b. Pratique I : Variables

En pratique, on utilise **ses variables** :

```
typedef struct s_ray
{
    double posx; //position x du joueur
    double posy; //position y du joueur
    double dirx; //vecteur de direction (commence à -1 pour N, 1 pour S, 0 sinon)
    double diry; //vecteur de direction (commence à -1 pour W, 1 pour E, 0 sinon)
    double planx; //vecteur du plan (commence à 0.66 pour E, -0.66 pour W, 0 sinon)
    double plany; //vecteur du plan (commence à 0.66 pour N, -0.66 pour S, 0 sinon)
    double raydirx; //calcul de direction x du rayon
    double raydiry; //calcul de direction y du rayon
    double camerax; //point x sur la plan camera : Gauche ecran = -1, milieu = 0,
                    droite = 1
    int mapx; // coordonnée x du carré dans lequel est pos
    int mapy; // coordonnée y du carré dans lequel est pos
    double sidedistx; //distance que le rayon parcours jusqu'au premier point
                      d'intersection vertical (=un coté x)
    double sidedisty; //distance que le rayon parcours jusqu'au premier point
                      d'intersection horizontal (= un coté y)
    double deltadistx; //distance que rayon parcours entre chaque point d'intersection
                       vertical
    double deltadisty; //distance que le rayon parcours entre chaque point
                       d'intersection horizontal
    int stepx; // -1 si doit sauter un carre dans direction x negative, 1 dans la
               direction x positive
    int stepy; // -1 si doit sauter un carre dans la direction y negative, 1 dans la
               direction y positive
    int hit; // 1 si un mur a ete touche, 0 sinon
    int side; // 0 si c'est un cote x qui est touche (vertical), 1 si un cote y
              (horizontal)
    double perpwalldist; // distance du joueur au mur
    int linewidth; //hauteur de la ligne a dessiner
    int drawstart; //position de debut ou il faut dessiner
    int drawend; //position de fin ou il faut dessiner
    int x; //permet de parcourir tous les rayons
};
```

#### c. Pratique II : Calculs

Pour chaque ray.x on va :

- **Calculer stepx, stepy, sidedistx et sidedisty.**
- **Incrémenter tant qu'on a pas touché un mur** (on passe au carré suivant soit dans la direction x soit dans la direction y). Ici, il faut savoir que l'on va d'abord jusqu'au

premier point d'intersection en parcourant une distance sidedistx et sidedisty. Puis on incrémente toujours de la même valeur, soit deltadistx et deltadisty.

- **Calculer perpwalldist pour avoir lineheight, puis drawstart et drawend.**

\*abs désigne la valeur absolue d'un nombre, c'est-à-dire sans prendre en compte son signe.

\*sqrt désigne la racine d'un nombre.

---

#### d. Pratique III : Dernières étapes

Il faut imprimer la colonne de pixel et adapter posx et posy aux mouvements droite/gauche et avancer/reculer.

---

## 2. Les Textures :

L'idée est ici de **récupérer la texture dans une image texture[0].img**. Puis de récupérer la couleur d'un pixel à (texx;texy) dans cette image afin de mettre la même couleur dans notre image de base data.img.

**Les textures doivent être au format xpm.** Celles-ci seront récupérées grâce à la fonction mlx\_xpm\_file\_to\_image.

**Attention :** Il faut penser à protéger la fonction si xpm est mauvais.

```
if (!(recup->texture[0].img = mlx_xpm_file_to_image(recup->datamlx_ptr,
    recuper->no, &(recup->texture[0].width), &(recup->texture[0].height))))
    ft_error(recup, "Texture SO\n");
```

L'objectif est de récupérer la couleur du pixel à (texx;texy) de la texture pour imprimer la même couleur dans notre image à (x;y). Il faut récupérer l'adresse de cette image dans le pointeur texture[0].addr avec mlx\_get\_data\_addr et calculer texx et texy.

```
texture[0].addr = (int *)mlx_get_data_addr(texture[0].img,
&texture[0].bits_per_pixel, &texture[0].line_length, &texture[0].endian);
data.addr[y * recuper->data.line_length / 4 + x] = texture[0].addr[texy *
texture[0].line_length / 4 + texx];
```

Le rôle des variables dans le code des textures est le suivant :

|        |  |
|--------|--|
| int    | texdir; //direction NO, S, EA, WE de la texture  |
| double | wallx; // valeur où le mur a été touché : coordonnée y si side == 0, coordonnée x si side == 1 |
| int    | texx; // coordonnée x de la texture  |
| int    | texy; // coordonée y de la texture   |
| double | step; // indique de combien augmenter les coordonnées de la texture pour chaque pixel          |
| double | texpos; // coordonnée de départ  |

### 3. Les Sprites :

```
void ft_header(t_recup *recup, int fd)
{
    int tmp;

    write(fd, "BM", 2); //La signature (sur 2 octets), indiquant qu'il s'agit d'un
                        //fichier BMP à l'aide des deux caractères.
                        // BM, 424D en hexadécimal, indique qu'il s'agit d'un
                        //Bitmap Windows.
    tmp = 14 + 40 + 4 * recuper->rx * recuper->ry; //La taille totale du fichier en
                                                    //octets (codée sur 4 octets)
    write(fd, &tmp, 4);
    tmp = 0;
    write(fd, &tmp, 2);
    write(fd, &tmp, 2);
    tmp = 54;
    write(fd, &tmp, 4);
    tmp = 40;
    write(fd, &tmp, 4);
    write(fd, &recup->rx, 4); //La largeur de l'image (sur 4 octets), c'est-à-dire le
                            //nombre de pixels horizontalement (en anglais
                            //width)
    write(fd, &recup->ry, 4); //La hauteur de l'image (sur 4 octets), c'est-à-dire le
                            //nombre de pixels verticalement (en anglais height)
    tmp = 1;
    write(fd, &tmp, 2); //Le nombre de plans (sur 2 octets). Cette valeur vaut
                        //toujours 1
    write(fd, &recup->data.bits_per_pixel, 2); //La profondeur de codage de la
                                                //couleur(sur 2 octets), c'est-à-dire
                                                //le nombre de bits utilisé
//pour coder la couleur. Cette valeur peut-être égale à 1, 4, 8, 16, 24 ou 32
    tmp = 0;
    write(fd, &tmp, 4); //La méthode de compression (sur 4 octets). Cette valeur
                        //vaut 0 lorsque l'image n'est pas compressée
    write(fd, &tmp, 4);
    write(fd, &tmp, 4);
    write(fd, &tmp, 4);
    write(fd, &tmp, 4);
    write(fd, &tmp, 4);
}
```

**Attention :** Pour que le --save fonctionne il faut qu'il passe dans les fonctions du raycasting mais qu'il existe directement après avoir la première vue.

---

#### **4. Derniers Petits Éléments :**

Pour quitter le programme proprement en appuyant sur la croix :

```
mlx_hook(recup->data mlx_win, 33, 1L << 17, ft_exit, recuper);
```

Si la taille de la fenêtre est supérieure à celle de l'écran, la taille de la fenêtre doit être celle de l'écran. Pour ça, utiliser la fonction `mlx_get_screen_size` :

```
mlx_get_screen_size(recup->data mlx_ptr, &recup->screenx, &recup->screeny);
recup->rx = (recup->rx > recuper->screenx) ? recuper->screenx : recuper->rx;
recup->ry = (recup->ry > recuper->screeny) ? recuper->screeny : recuper->ry;
```

---

#### **5. Les leaks :**

Face au leaks : utiliser `-fsanitize=leak` et `valgrind`.

Pour déterminer si le leaks provient de la MiniLibX ou de soit, exécuter : `valgrind --leak-check=full --show-leak-kinds=all ./executable description.cub` (la technique de rajouter `2> leak.log` pour que tous les leaks soit dans un fichier).

Pour plus de lisibilité, utiliser `valgrind --leak-check=full --show-leak-kinds=all ./executable description.cub 2> leak.log`.

**Pour free quelque chose, utiliser la condition if(str) existe.** De fait, il faut initialiser les valeurs que l'on free.

**Faire attention à ne pas free deux fois, free sans malloquer, free sans initialiser et écrire des pixels en dehors de l'image.**

**Le malloc doit être free même lorsque qu'il y a une erreur, le --save ou s'il y a une erreur de malloc.**

---

**FIN**