Cub3D
Code explanation

Helpful resources:
https://lodev.org/cgtutor/raycasting.html
https://ismailassil.medium.com/ray-casting-c-8bfae2c2fc13
https://www.youtube.com/watch?v=NbSee-XM7WA

I) Parsing

Parse_args() first checks whether the number of arguments is 2 (the first being the program's name and the second the map file name) the extension of the second argument has the correct extension type: .cub

Parse_file() creates a data struct and initiates its variables. It then opens the map file and deals with it line by line thanks to get_next_line(). Each line is processed by process_line() and freed. At the end of the file, the data needs to be checked for accuracy, by data_check(), and the map file is closed.

Process_line():

It ignores spaces at the beginning of the current line.

Then if the line is empty (by meeting a \n), either we ignore it by simply returning if that line is not inside the map, otherwise we set a variable in_map, to indicate that we are inside the map. Indeed, if another map line is found after that empty line, that would mean a wrong map.

If an orientation is met (is_orientation() tells us if we met "NO ", "SO ", "WE " or "EA "), then we send the line starting after those characters to parse_texture(). Parse_texture needs to know if it deals with north, south, west or east, and it gets this argument from get_option().

If "F " (floor) or "C " (ceiling) is met, then we send the line to parse_color().

If none of those options are verified, it means we are dealing with the map itself. However, the map needs to be after its description in the file: to check that, we use end_of_map().

Parse_texture() first replaces the final '\n' to '\0', then any character that is not the g of .png is replaced by '\0'. That will allow us to deal with any final spaces. Then we increment the index to pass any spaces.

Texture_duplicate_check() whether the current wall has already been associated with a texture. If it has, it means the current line is a duplicate of that orientation.

If not, we attribute to the corresponding string in data the path of the texture file with full_texture_path().

Parse_color(), like the texture function, ignores any spaces at the beginning of the line. Then we check for each character if it is valid. If the test is passed, parse_color_helper() fills the corresponding string in data (floor or ceiling) and the rgb (Red Green Blue) integer with get_rgb(). This function checks with check_number() if we are dealing with a positive number (either the string starts with a digit, or with a + followed by a digit). Then we use ft_atoi() to get r, g and b. Afterwards, we check if each integer is between 0 and 255 (RGB cannot exceed those limits). Finally, we return the whole RGB integer by shifting bits:

(r << 16): moves the red component 16 bits to the left : RR0000. (g << 8): moves the green component 8 bits to the left: 00GG00. b stays at its place since it needs to be at the far right: 0000BB. The | combines everything: RRGGBB.

Parse_map_line() checks if the variable for "empty line after map line" is set, in which case we return an error.

Then, we check for tabs and send back an error. Indeed, a tab can mess up the alignment of the characters in the map: one tab can have different sizes depending on the software used to read the file, and it corresponds to only one character '\t'... So we just avoid it altogether.

Then, we verify the validity of every single character in the map. The player is represented by its orientation: N, S, W or E. There can only be one player.

Finally, realloc_line() stores the whole map in one string in the data structure.

This marks the end of the process_line() function.

In **data_check()**, we verify the presence in the data structure of every texture, color, and the map (if the map was not at the end of the file, then it was never parsed by process_line()!). We make sure there is a player, and finally the function wall_outline() will check for walls.

Wall_outline() splits the map string into a string's array. The first_tab_line() and last_tab_line() function make sure that those are either 1 (wall) or space. The middle_tab_lines() goes through every middle line and if it encounters a 0 or a space, the space() or the zero() functions will, respectively, make sure that the characters on top of it, below it, on its right and on its left, are acceptable characters. If not, it means there is a hole in the map.

II) Initiation of game

Init_game() initiates a connection to the MLX, providing the size of the window (defined in the header file: 1024*720). A game structure is created and is connected to the data structure, the map array in the data structure, a future image, and textures through load_textures().

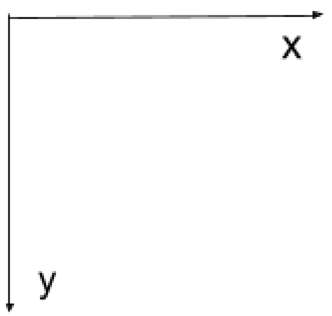
Load_textures() creates an array of textures (the tex type is actually mlx_texture_t, for better readability we defined a macro in the header file). In this array, [0] is the north texture, [1] the south, [2] the west and [3] the east. We use the mlx' function mlx_load_png() with, as argument, the string of the path to the corresponding texture.

III) Rendering initiation

Init_render_data() sets:
- the first position of the player with set_starting_position(),
- the direction he is looking in with set_starting_direction(),
- the plane camera vue with set_camera_plane()
- the timer with init_time().

Set_starting_position() needs the size of the map. Get_map_width() and get_map_height() provide that information by "measuring" the longest row and the longest column. Then, browsing all coordinates x and y, if we encounter the player, we set its position (x, y) adding to it 0,5: indeed, we want him to be in the "middle" of the box and not stuck to a side. If the player is next to a wall, it would be a weird view if he was at a distance of 0 of the wall. Those coordinates are stored in a render_data structure that was created in the main (but not yet filled).

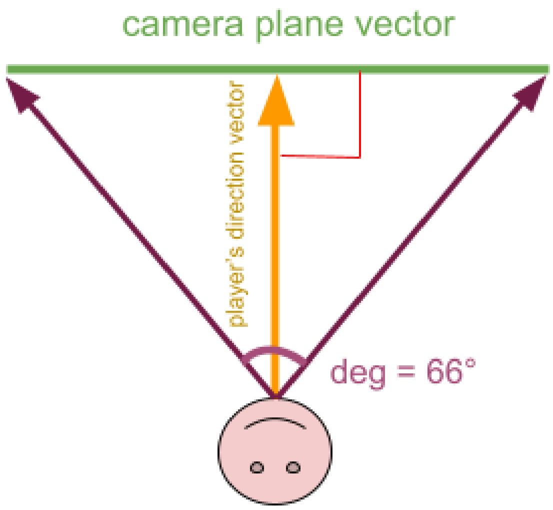
Set_starting_direction() sets the dir_x and dir_y coordinates to describe the orientation of the player (north, south, west, east). Our (x, y) plane is as follows:



Y is facing down because this is the way we go through the map's array, incrementing rows and columns.

Therefore, if the orientation is north, our direction vector (x, y) will be (0, -1). If it is south, it will be (0, 1). West is (-1, 0), east is (1, 0).

Set_camera_plane() defines a FOV (Field Of View) : (plane_x, plane_y). The camera plane is a vector that is perpendicular to the player's direction vector, and its length determines the vision: the smaller the vector, the more it will appear zoomed in.

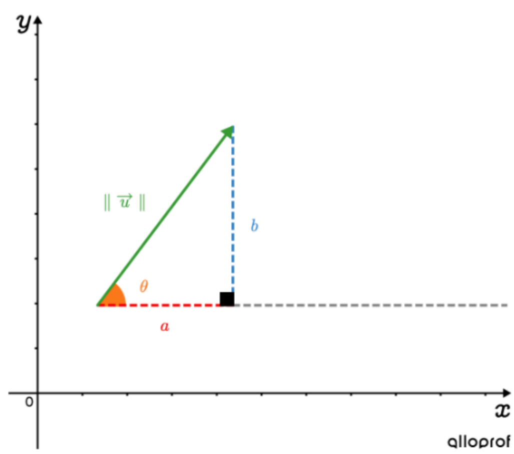


The angle of view fov_deg is fixed to 66 (like in the Wolfenstein game).

Fov_rad is the conversion in radian of the degree, this is necessary to get the length of the camera view with tan. M_PI is pi = 3,14…

As a reminder:

$$\tan \theta = \frac{b}{a}$$



With the player's direction vector and the length of the camera's plane, we extract the coordinates of the camera's plane, and store them in the render_data structure.

Finally, **init_time()** sets to 0 two timers: time and old_time. We will see why later.

This marks the end of **init_render_data()**.

IV) Mlx key management

Mlx_key_hook() is a MLX function. With my_keyhook(), if the escape key is pressed, the function my_mlx_close() is called. This way, we can close the window properly.

Mlx_close_hook() is also a MLX function. It corresponds to the event of clicking the X button of the window. Same as before, the my_mlx_close() function is called to close everything properly.

V) Rendering loop

Mlx_loop_hook() is a MLX function that creates a loop of the program, by calling the render_loop() function.

Render_loop() creates a t_ray_data structure.
If no image was previously created, then we create one with mlx_new_image() which is a MLX function. Then the image is put on the window by mlx_image_to_window().
The current time is set by mlx_get_time().
The frame_time is the difference between the current and the old time.
The old time is the current time.
The move_speed and the rot_speed depend on the frame_time, for smooth movements of the player and rotation view.
Update_keys() manages the movements of the player and rotation of the view.

Clear_image() sets all the pixels of the window in black with full opacity.

Draw_ceiling() sets the pixels of the upper half of the window with my_mlx_pixel_put() to the correct color (parsed in RGB previously).

Draw_floor() does the same for the lower half of the window with the floor color.

Raycast() is the actual raycasting function, we will go into it further later.

Let’s go into detail with **update_keys()**:

Move_forward() creates new coordinates x and y for the position of the player. These new coordinates are obviously in the direction of the direction vector, since we want to move forward (forward = in the direction we are looking in). The speed of movement is a multiplier: the higher the speed, the further away the new coordinate. If the W key is pressed (that condition is checked with mlx_is_key_down()), then we set the new position of the player, as long as the new coordinates are not a wall (a 1 in the map’s array).

Move_backward() operates in the same way but the direction is negative (we want to move along the direction’s vector, but the opposite way).

Move_right() and **move_left()** use the camera’s plane vector as a reference as opposed to the direction vector. Since we set the plane’s vector as pointing to the right, it is negated in move_left().

Rotate_right() and **rotate_left()** use cosinus and sinus operations to change the orientation of the direction vector. This stands from the following trigonometry notion:

$$\begin{cases} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta \end{cases}$$

Push_player() manages collisions with the walls. We use a small margin of 0,1 (COLL_RAD) to avoid strange visuals. If the position of the player coincides with a wall, then we shift the player in the opposite way (in a distance of 0,1), but only after checking if there is no wall in that opposite direction. In a nutshell, if there is a wall immediately on the right, we check if there is no wall immediately on the left, and we shift the player to the left.

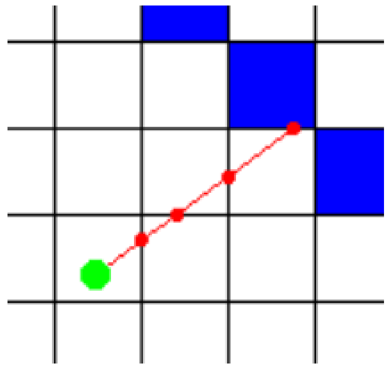
VI) Raycasting

Raycast() browses every vertical line of the image: x goes through the whole width of it:

Calc_ray_direction() calculates the direction of the ray for that vertical line (x). The position of that line (camera_x) on the screen is set relatively to the width of the image.

Set_map_cell() assigns the player's coordinates to a box in the map (casting to integer because we have grid-based map, meaning you cannot have a half-box, so no decimals).

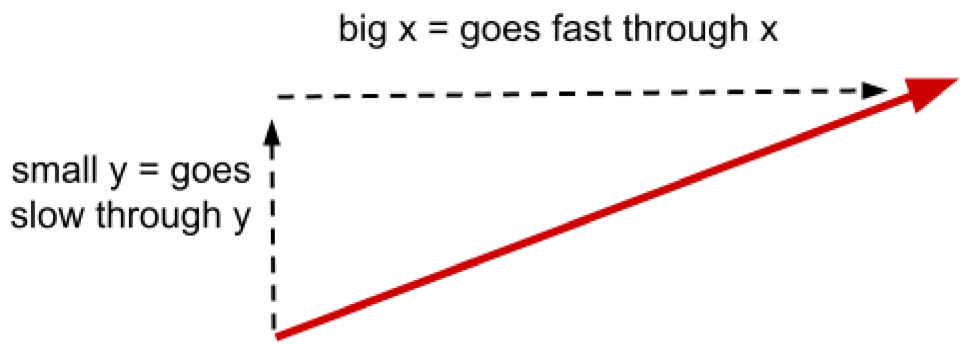
Set_delta_dist() calculates the distances the ray has go to cross the x and y of a box.



If the x direction component of the vector is 0, it means the ray is vertical; we set a very long distance. Same for y = 0 which is a horizontal ray.

Otherwise the ray is diagonal: x and y both are != 0. In that case, the distance is calculated by 1/ray_direction_x or y.

For instance, if the x component of the ray vector is high, then the vector goes "fast" in the x direction, and the distance it goes before crossing the box' x limit will be smaller (thus inverting to 1/ray_direction_x). The math.h's function fabs() ("floating-point absolute value") returns the absolute value of a float/double/long double type value (which has decimals).



Calculate_step() calculates the direction the ray has to go on the x and y axis. It is negative if the ray goes in the opposite direction of the axis (so x < 0 if it is going to the left, and y < 0 if it is going upwards). Side_dist_x and y calculate the initial distances between the ray and the borders of the box (x and y respectively). It uses the delta_dist calculated previously.

Dda() implements the Digital Differential Analyzer algorithm. It determines where a ray touches a wall in a grid map. Hit is a variable that signals if a wall was hit. If the distance on the x axis is shorter than y, then the ray will cross the x side of the box. If a box' side is crossed, then we readjust side_dist. If the side is a wall (1), hit is set to 1.
Set_wall_direction() will tell us if the wall in question is north, west, east or south.

Set_perp_wall_dist() calculates the perpendicular distance between the player and the wall that's touched by the ray. This distance is necessary to render the perspective properly. Data->side indicates whether the wall is horizontal (1) or vertical (0). Side_dist_x and y store the distance until the box' border that induced the collision. In the last DDA iteration, the delta was added one too many times. So we subtract it to correct that. If the distance is < 0,1, then it is too small for the calculation, so we set it to 0,1.

Set_line_height() calculates the height of the line that represents the wall. Each line is a column of pixels in the 3D view. To get the line's height, we divide the image height by the perpendicular wall distance. The farther the wall, the shorter the line (and therefore the shorter the wall appears to be). Draw_start and draw_end are the top and the bottom of the wall’s drawing. We place the drawing of the wall at the center of the screen. Otherwise the walls would be drawn too low or too high depending on the distance.

```
+--------------------+  ← y = 0 (haut de l'écran)
|                    |
|       Ciel         |
|    --------------  |  ← y = draw_start (début du mur)
|    |    Mur     |  |
|    |            |  |
|    --------------  |  ← y = draw_end (fin du mur)
|       Sol          |
|                    |
+--------------------+  ← y = height - 1 (bas de l'écran)
```

Set_wall_x() calculates the horizontal position of the impact on the wall. That is needed for texture application. Floor() extracts the integer part of the coordinate, leaving only the decimal. What’s left represents the relative position of the impact point on the surface of the wall: 0.0 being an impact on the left side of the wall, 1.0 on the right side. This allows for proper mapping of the texture on the wall, depending on where the ray touches it.

Set_texture_index() assigns an index to the direction of the wall. This will allow us to choose the correct texture.

Draw_texture_stripe() draws a vertical stripe (x) of the texture image, depending on the wall detected by the raycasting. First we catch the right texture from the index set previously. Then we multiply wall_x by texture_width to get the correct column. We check the limits: we need to stay within the texture limits, to avoid memory bogs. The step corresponds to how many pixels we pass with one stripe: how many pixels of the texture correspond to one pixel of the screen. Draw_end - start is the height of the wall on the screen. We then divide texture->height by the height of the wall on screen, in order to put the texture to the scale of the wall. Then we calculate the initial position of the texture: we center it vertically (draw_start - height / 2), move it to be well aligned ((draw_end - start)/2), and we convert the screen unit of measurement into the texture unit by multiplying by step.

Stripe_loop() draws the texture for each pixel of the column. It browses the texture image and puts it on the wall that appears on screen, taking into account the position and perspective. Tex_pos is the vertical position in the texture; we use & (texture->height - 1) to stay within the limits of the texture (to avoid going too high): indeed, if the height of the texture is 42, then &(42-1) = &41 is the same as operating a modulo 41. Texture_pixels is an array of colors, organised pixel by pixel. Tex_x and y are the coordinates of the pixel in the texture that we need to draw. We get the index of the pixel in the array with (data->tex_y * texture->width + data->tex_x). Each pixel is stored in 4 bytes for RGBA, so we multiply by 4. *((uint32_t *)...) recovers the pixel's color in RGBA format. My_pixel_put_texture places the color on the screen at x, y. Finally, by incrementing y, we go to the pixel of the next row of the vertical line.

However, because we are dealing with .png for the texture, the RGBA actually needs to be BGRA (blue, green, red, opacity), so we need to shift bits again.

That marks the end of the raycast() function.

VII) Termination

Mlx_terminate() is a MLX42 function that terminates the game.

My_mlx_close() is a function that closes and frees every variable and memory needed to properly exit the simulation.

Delete_texture_tab() uses the MLX42 function mlx_delete_texture() to get rid of the textures, and frees the array.

Mlx_close_window() closes the window.

Mlx_delete_image() deletes the image.

Mlx_terminate() terminates the connection with the MLX library.

Cleanup() frees every variable in the data structure, closes the file descriptor if necessary and frees the structure itself.

Finally, we **exit(0)** for success.

VIII) Minimap

The minimap is a bonus that we chose to implement. A small map representing all or part of the maze appears on the bottom left of the screen. The player is represented by a red square, and a line represents the direction he's facing.

First, the minimap is initialized by **init_minimap_data()** in the main: we measure the width and height of the actual map, and decide that the minimap will be 200 by 200 pixels. The tile represents a box of the map: 0 or 1. The tile size depends on the number of boxes in the map. We adjust the size to be a bit bigger by a factor of 1,5 (so 50% bigger). Therefore, we won't be able to render every tile, so we calculate how many we will see, by dividing the size of the map by the size of a tile.

In the render_loop() function, the minimap image is created through mlx_new_image(), and pushed to the window by images_to_window().

After the raycast() function, we **draw_minimap()**.

Update_offset() makes sure the player is centered in the minimap with every movement, while also making sure the minimap cannot get out of the limits of the map.

Draw_walls() puts white or black pixels depending on if it is a wall or not.

First, we shift the coordinates to center the minimap. We convert the pixels into real coordinates. So we get the coordinates on the map associated with the pixel. Floor() allows to round up to the correct box of the map.

For each pair of coordinates, if it's a wall (1), we put a white pixel. Otherwise, we put a black one.

Draw_player() draws a red square for the player and a line for his direction.

Pos_x and pos_y are his real position in the map. We shift it with the previously calculated offset, to center it. We convert the coordinates by multiplying by the tile size.

Then we calculate x1, y1 coordinates to be the end of the direction line.

The player is drawn as a square that’s the size of one third of the tile.