cub3D

My first RayCaster with miniLibX

Summary: This project is inspired by the world-famous Wolfenstein 3D game, which was the first FPS ever. It will enable you to explore ray-casting. Your goal will be to make a dynamic view inside a maze, in which you'll have to find your way.

Version: 10

Contents

Chapter I

Foreword

Developed by `Id Software` by the über famous John Carmack and John Romero, published in 1992 by `Apogee Software`, `Wolfenstein 3D` is the first true "First Person Shooter" in the history of video games.

Figure I.1: John Romero (left) and John Carmack (right) posing for posterity.

`Wolfenstein 3D` is the ancestor of games like `Doom` (`Id Software`, 1993), `Doom II` (`Id Software`, 1994), `Duke Nukem 3D` (`3D Realm`, 1996) and `Quake` (`Id Software`, 1996), that are additional eternal milestones in the world of video games.

Now, it's your turn to relive History...

Chapter II

Goals

This project's objectives are similar to all this first year's objectives: Rigor, use of `C`, use of basic algorithms, information research etc.

As a graphic design project, `cub3D` will enable you to improve your skills in these areas: windows, colors, events, fill shapes, etc.

To conclude `cub3D` is a remarkable playground to explore the playful practical applications of mathematics without having to understand the specifics.

With the help of the numerous documents available on the internet, you will use mathematics as a tool to create elegant and efficient algorithms.

We recommend that you test the original game before starting this project:
http://users.atw.hu/wolf3d/

Chapter III

Common Instructions

- Your project must be written in C.
- Your project must be written in accordance with the Norm. If you have bonus files/functions, they are included in the norm check and you will receive a `0` if there is a norm error inside.
- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc) apart from undefined behaviors. If this happens, your project will be considered non functional and will receive a `0` during the evaluation.
- All heap allocated memory space must be properly freed when necessary. No leaks will be tolerated.
- If the subject requires it, you must submit a `Makefile` which will compile your source files to the required output with the flags `-Wall`, `-Wextra` and `-Werror`, use cc, and your Makefile must not relink.
- Your `Makefile` must at least contain the rules `$(NAME)`, `all`, `clean`, `fclean` and `re`.
- To turn in bonuses to your project, you must include a rule `bonus` to your Makefile, which will add all the various headers, librairies or functions that are forbidden on the main part of the project. Bonuses must be in a different file `_bonus.{c/h}` if the subject does not specify anything else. Mandatory and bonus part evaluation is done separately.
- If your project allows you to use your `libft`, you must copy its sources and its associated `Makefile` in a `libft` folder with its associated Makefile. Your project's `Makefile` must compile the library by using its `Makefile`, then compile the project.
- We encourage you to create test programs for your project even though this work **won't have to be submitted and won't be graded**. It will give you a chance to easily test your work and your peers' work. You will find those tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to your assigned git repository. Only the work in the git repository will be graded. If Deepthought is assigned to grade your work, it will be done

after your peer-evaluations. If an error happens in any section of your work during Deepthought's grading, the evaluation will stop.

Chapter IV

Mandatory part - cub3D

Program name	`cub3D`
Turn in files	`All your files`
Makefile	`all, clean, fclean, re, bonus`
Arguments	`a map in format *.cub`
External functs.	• `open, close, read, write, printf, malloc, free, perror, strerror, exit` • `All functions of the math library (-lm man man 3 math)` • `All functions of the MinilibX`
Libft authorized	`Yes`
Description	`You must create a "realistic" 3D graphical representation of the inside of a maze from a first-person perspective. You have to create this representation using the Ray-Casting principles mentioned earlier.`

The constraints are as follows:

- You `must` use the `miniLibX`. Either the version that is available on the operating system, or from its sources. If you choose to work with the sources, you will need to apply the same rules for your `libft` as those written above in `Common Instructions` part.
- The management of your window must remain smooth: changing to another window, minimizing, etc.
- Display different wall textures (the choice is yours) that vary depending on which side the wall is facing (North, South, East, West).

- Your program must be able to set the floor and ceiling colors to two different ones.
- The program displays the image in a window and respects the following rules:
  - The left and right arrow keys of the keyboard must allow you to look left and right in the maze.
  - The W, A, S, and D keys must allow you to move the point of view through the maze.
  - Pressing `ESC` must close the window and quit the program cleanly.
  - Clicking on the red cross on the window's frame must close the window and quit the program cleanly.
  - The use of `images` of the `minilibX` is strongly recommended.
- Your program must take as a first argument a scene description file with the `.cub` extension.
  - The map must be composed of only 6 possible characters: **0** for an empty space, **1** for a wall, and **N**,**S**,**E** or **W** for the player's start position and spawning orientation.
    This is a simple valid map:

```
111111
100101
101001
1100N1
111111
```

  - The map must be closed/surrounded by walls, if not the program must return an error.
  - Except for the map content, each type of element can be separated by one or more empty line(s).
  - Except for the map content which always has to be the last, each type of element can be set in any order in the file.
  - Except for the map, each type of information from an element can be separated by one or more space(s).
  - The map must be parsed as it looks in the file. Spaces are a valid part of the map and are up to you to handle. You must be able to parse any kind of map, as long as it respects the rules of the map.

- Each element (except the map) firsts information is the type identifier (composed by one or two character(s)), followed by all specific informations for each object in a strict order such as :
  - North texture:

```
NO ./path_to_the_north_texture
```

    - identifier: **NO**
    - path to the north texure
  - South texture:

```
SO ./path_to_the_south_texture
```

    - identifier: **SO**
    - path to the south texure
  - West texture:

```
WE ./path_to_the_west_texture
```

    - identifier: **WE**
    - path to the west texure
  - East texture:

```
EA ./path_to_the_east_texture
```

    - identifier: **EA**
    - path to the east texure
  - Floor color:

```
F 220,100,0
```

    - identifier: **F**
    - R,G,B colors in range [0,255]: **0, 255, 255**

- Ceiling color:

```
C 225,30,0
```

  - identifier: **C**
  - R,G,B colors in range [0,255]: **0, 255, 255**

- Example of the mandatory part with a minimalist **.cub** scene:

```
NO ./path_to_the_north_texture
SO ./path_to_the_south_texture
WE ./path_to_the_west_texture
EA ./path_to_the_east_texture

F 220,100,0
C 225,30,0

        1111111111111111111111111
        1000000000110000000000001
        1011000001110000000000001
        1001000000000000000000001
111111111011000001110000000000001
100000000011000001110111111111111
11110111111111011100000010001
11110111111111011101010010001
11000000110101011100000010001
10000000000000001100000010001
10000000000000001101010010001
11000001110101011111011110N0111
11110111 1110101 101111010001
11111111 1111111 111111111111
```

- If any misconfiguration of any kind is encountered in the file, the program must exit properly and return "Error\n" followed by an explicit error message of your choice.

Chapter V

Bonus part

Bonuses will be evaluated only if your mandatory part is PERFECT. By PERFECT we naturally mean that it needs to be complete, that it cannot fail, even in cases of nasty mistakes like wrong uses etc. It means that if your mandatory part does not obtain ALL the points during the grading, your bonuses will be entirely IGNORED.

Bonus list:

- Wall collisions.
- A minimap system.
- Doors which can open and close.
- animated sprite.
- Rotate the point of view with the mouse.

You will be able to create better games later do not waste too much time!

You are allowed to use other functions or add symbols on the map to complete the bonus part as long as their use is justified during your evaluation. You are also allowed to modify the expected scene file format to fit your needs. Be smart!

Chapter VI

Examples

Figure VI.1: `Wolfeinstein3D` original game usinfg RayCasting.

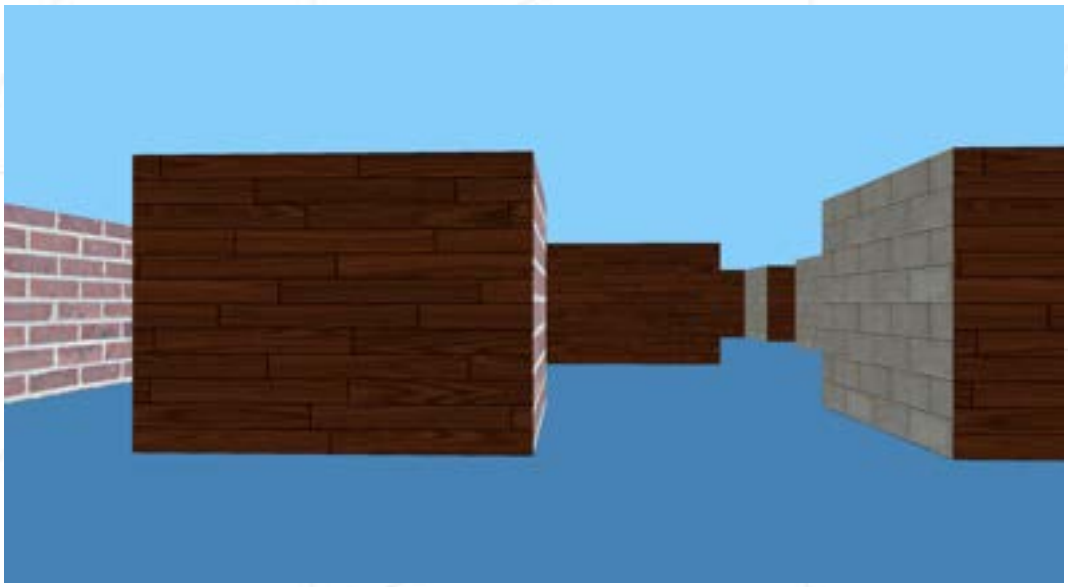

Figure VI.2: Example of what your project could look like as per the mandatory part.

Figure VI.3: Example of the bonus part with a minimap, floor and ceiling textures and an animated famous hedgehog sprite.

Figure VI.4: Another example of a bonus with a HUD, health bar, shadow effect and weapon that can shoot

Figure VI.5: Another example of a bonus game with a weapon of your choice and the player looking at the ceiling

Chapter VII

Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your files to ensure they are correct.

```
???????????  XXXXXXXXXX = $3$$796ba5a53df1352e06cc7b0f3ad2a41d
```