Cub3d : Ray-Casting

Assignment

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 :

• 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.

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.

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 texture
* South texture:
SO ./path_to_the_south_texture
· identifier: SO
· path to the south texture
* 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
111111111111111111111111
100000000011000000000001
101100000111000000000001
100100000000000000000001
11111111101100000111000000000001
10000000001100000111011111111111
11110111111111011100000010001
11110111111111011101010010001
11000000110101011100000010001
10000000000000001100000010001
10000000000000001101010010001
11000001110101011111011110N0111
11110111 1110101 101111010001
11111111 1111111 111111111111
```

Install MLX42

It appears that the README content got cut off, but based on what's there, I can give you some guidance on how to compile and link MLX42 with your C program on MacOS.

1. **Install the Dependencies**: If you haven't already installed the dependencies such as GLFW and CMake, you can use Homebrew to install them.

   ```
   brew install glfw
   brew install cmake
   ```

2. **Clone and Build MLX42**: You should have already cloned MLX42. To build it, navigate to the directory and execute:

   ```
   cd MLX42
   cmake -B build
   cmake --build build -j4
   ```

   This will generate `libmlx42.a` in the `build` directory.

3. **Compile Your Program**: To compile your program along with the MLX42 library, you would typically run:

   ```
   gcc main.c libmlx42.a -Iinclude -framework Cocoa -framework OpenGL -framework IOKit -lglfw
   ```

   Note: Replace `main.c` with the name of your source file and include any additional source files and header files as required.

   In the case that your glfw library is located in a custom directory, make sure to specify it using the `-L` flag like so:

   ```
   gcc main.c libmlx42.a -Iinclude -framework Cocoa -framework OpenGL -framework IOKit -L"/Users/$(USER)/.brew/o
   pt/glfw/lib/" -lglfw
   ```

4. **Error Handling**: When writing your code, it's advisable to handle errors extensively. Make use of functions that return error codes or null pointers to check if something has gone wrong, and then take the appropriate action, such as logging an error message or cleaning up resources.
5. **Execute the Program**: After successfully compiling, run the output executable.
6. **MacOS Security**: If MacOS complains about security, navigate to `Settings > Security & Privacy` and allow the application to run.

Doku and Tutorials

Tutorial

https://lodev.org/cgtutor/raycasting3.html

Github

https://github.com/l-yohai/cub3d/blob/master/mlx_example/01_untextured_raycast.c

Parsing

Steps to Implement Parsing

1. **File Reading**
2. **Line-by-Line Parsing**
3. **Validating Map Rules**
4. **Putting it All Together**

1 - File reading

```
//first open the .cub file using open()
#include <stdio.h>
#include <stdlib.h>

FILE *open_config_file(const char *path)
{
    FILE *fd = fopen(path, "r");
    if (!fp){
        perror("Error");
        exit(1);}
    return fd;
}
```

2- Line-by-Line parsing

Parse each line according to the identifiers ("NO", "SO", "WE", "EA", "F", "C").

```
void parse_line(char *line)
{
    if (line[0] == 'N' && line[1] == 'O')
    {
        // Parse North Texture path
    }
    else if (line[0] == 'S' && line[1] == 'O')
    {
        // Parse South Texture path
```

```
    }
    else if (line[0] == 'W' && line[1] == 'E')
    {
        // Parse West Texture path
    }
    else if (line[0] == 'E' && line[1] == 'A')
    {
        // Parse East Texture path
    }
    else if (line[0] == 'F')
    {
        // Parse Floor color
    }
    else if (line[0] == 'C')
    {
        // Parse Ceiling color
    }
    else
    {
        // Parse the map here
    }
}
```

2a - Parse the textures with DRY method

2b - Dynamically allocate and fill the 2D array `map` inside the `parse_line()` function and count the `rows` and `cols` as you went along.

```
// a function to initialize the structure t_maprqs
t_mapreqs init_mapreqs(void)
{
  t_mapreqs mapreqs;

  mapreqs.pos_x = 0;
  mapreqs.pos_y = 0;
  mapreqs.space = 0;
  mapreqs.wall = 0;
  mapreqs.orientation = 0;
  return (mapreqs);
}

int is_map_line(char *line)
{
  int i;

  i = 0;
  if (line == NULL)
    return (false);
  printf("Debug: line = '%s'\n", line);  // Debug print
  while (line[i] && is_space(line[i]))
    i++;
  if (line[i] == '\0')
    return (true);
  while (line[i])
  {
    if (!ft_strchr(" 01NESW", line[i++]))
      return (false);
```

```
  }
  return (true);
}

t_map *init_map_dimensions(t_map *map_i, int fd)
{
  map_i->height = 0;
  map_i->width = 0;
  map_i->line = get_next_line(fd);

  while (map_i->line && is_map_line(map_i->line))
  {
    map_i->height++;
    map_i->width = ft_strlen(map_i->line);
    free(map_i->line);
    map_i->line = get_next_line(fd);
  }
  if (map_i->line)
    free(map_i->line);
  return (map_i);
}

t_map *allocate_map_memory(t_map *map_i)
{
  map_i->tiles = malloc(sizeof(char *) * (map_i->height + 1));
  if (!map_i->tiles)
    ft_error("Could not allocate memory");
  return (map_i);
}

// fill the map with the map file, units are characters,
// space is allowed and is a valid part of the map.
t_map *fill_map_tiles(t_map *map_i, int fd)
{
  int i;

  i = 0;
  map_i->tiles[i] = get_next_line(fd);
  while (map_i->tiles[i] && is_map_line(map_i->tiles[i]) && i < map_i->height)
  {
    map_i->tiles[i] = get_next_line(fd);
    i++;
  }
  return (map_i);
}


t_map init_map(char *map, t_map *map_i)
{
  int fd;

  map_i->mapreqs = init_mapreqs();
  fd = open_file(map);
  map_i = init_map_dimensions(map_i, fd);
  if (!map_i)
  {
    close(fd);
    free_map(map_i, "Could not initialize map dimensions");
    exit(1);
  }
  map_i = allocate_map_memory(map_i);
  if (!map_i)
  {
    close(fd);
    free_map(map_i,"Could not allocate memory for map");
```

```
    exit(1);
  }
  // fd = open(map, O_RDONLY);
  // if (fd < 0)
  //  free_map(map_i, "Could not open the map file");
  map_i = fill_map_tiles(map_i, fd);
  printf("Debug: HI\n");  // Debug print
  if (close(fd) < 0){
    free_map(map_i, "Could not close the map file");
    exit(1);
  }
  return (*map_i);
}
```

3- Validating Map Rules

```
void validate_map(char **map, int rows, int cols)
{
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            // Check if map is surrounded by walls and other validations.
        }
    }
}
```

4- Putting it All Together

```
#include <stdio.h>
#include <stdlib.h>

FILE *open_config_file(const char *path);
void parse_line(char *line);
void validate_map(char **map, int rows, int cols);

int main(int argc, char **argv)
{
    if (argc != 2)
    {
        fprintf(stderr, "Error: Invalid number of arguments\n");
        return 1;
    }

    FILE *fp = open_config_file(argv[1]);
    char *line = NULL;
    size_t len = 0;

    char **map = NULL;
// You'll dynamically allocate and fill this as you parse
    int rows = 0;
    int cols = 0; // Assuming a rectangular map, otherwise, you may need an array to keep track of cols for each row

    while (getline(&line, &len, fp) != -1)
    {
        parse_line(line);
        // Here you would update `map`, `rows`, and `cols` based on the parsed line
```

```
    }

    // Validate the map after all lines are parsed and map is populated
    validate_map(map, rows, cols);

    free(line);
    fclose(fp);
    return 0;
}

FILE *open_config_file(const char *path)
{
    // ... (same as before)
}

void parse_line(char *line)
{
    // ... (same as before)
    // Here you would update `map`, `rows`, and `cols` based on the parsed line
}

void validate_map(char **map, int rows, int cols)
{
    // ... (same as before)
}
```

Unit Testing of Parsing

Certainly, unit tests can help you ensure that each part of your project is working as expected. For a C project, frameworks like Check, CUnit, or Unity can be useful, but I'll use basic assertions for simplicity here. Each function that performs a piece of logic should have a corresponding unit test that verifies its correctness.

Here are some example unit tests for different aspects:

1. Test for `parse_north_texture()`

```
#include <assert.h>
#include <string.h>

void test_parse_north_texture() {
    t_config config;
    char line[] = "NO ./path/to/the/north_texture";
    parse_north_texture(line, &config);

    assert(strcmp(config.north_texture, "./path/to/the/north_texture") == 0);

    free(config.north_texture);
}
```

2. Test for `open_config_file()`

Here, you would create a dummy `.cub` file with some simple content and try to open it.

```
void test_open_config_file() {
    FILE *fp = open_config_file("./dummy.cub");
    assert(fp != NULL);
```

```
    fclose(fp);
}
```

3. Test for `validate_map()`

```
void test_validate_map() {
    char *map[] = {
        "111111",
        "100101",
        "101001",
        "1100N1",
        "111111",
    };

    assert(validate_map(map, 5, 6) == 1); // 1 means valid, 0 means invalid
}
```

Putting it All Together (main function)

```
int main() {
    test_parse_north_texture();
    test_open_config_file();
    test_validate_map();
    printf("All tests passed!\\n");
    return 0;
}
```

Each test function sets up a scenario using either real data or mock data, then runs the function you want to test, and uses assertions to confirm that the function behaved as expected. You can run these test functions before your `main()` to check if everything is working as intended.

Note that these are simplified unit tests and are only meant to give you an idea of how to write them. You may also want to add more tests for boundary cases, invalid inputs, and so on.

slakner cub