ECOLE
POLYTECHNIQUE
DE BRUXELLES

INFO-H502 - Virtual Reality

Project report

3D Maze

Authors :

Yassine Ben Yaghlane

Rania Charkaoui

Arthur Elskens

Academic year 2021-2022

Contents

1 3D Maze game

In order to get out of the maze, the player has to open a door. The door will only open if one has gathered all the necessary keys and has to be found by the player.

The player can move in all directions, and jump. The controls allowed are:

- **ArrowUp**: it moves the cube forward, in the direction of the rolling_front vector which always points in front of the cube
- **ArrowDown**: it is the opposite of ArrowUp
- **ArrowRight**: it rotates clockwise the cube around the rolling_up vector which always points in the direction of the y-axis;
- **ArrowLeft**: it rotates anti-clockwise the cube along the same axis as ArrowRight
- **D**: it moves the cube in the direction of the rolling_right vector which always points orthogonally to the right of rolling_front;
- **Q**: it is the opposite of ArrowRight; Combination of the first four controls (which are not opposed movements) are possible.
- **R**: it allows the player to reset and be placed at the starting position if he/she is lost.
- **Space bar**: it allows to jump in the Y-axis

All the movements can be combined.

The game is structured with the Object Oriented Paradigm. Each object in the game is a GameObject and all the relevant classes inherit from this class.

The video can be found at: https://youtu.be/mXoZKzcaz_k. The source code can be found at :https://github.com/YassineBenYaghlane/WebGLMaze

2 Classic functionalities

2.1 Game Logic

The maze is designed with walls, doors, ceiling, floor and key objects. Their location is based on a .txt file (see Figure 1) which is parsed and for each character correspond a specific object to create and rendered. There are 2 different maps with 2 different layouts. Each object is originally a cube object except for the keys, scaled to desired look. Each of them also has a texture.

In order to get out of the maze through the doors, one has to gather 3 keys per door and find the doors. The number of keys detained by the player is displayed on the screen and is in its inventory.

The player can also decide to get back to the start of the maze with the *reset* button, the 'R' key, if he/she is lost.

When the player gets to a door with 3 keys, there is an animation of the door sliding to let the player in.

Once the first door is open, the player can enter the second darker part of the maze where the principle remains the same as before, finding the keys to escape.

Figure 1: Example of a map.txt

2.2 Shaders

Four pairs of vertex and fragment shader are used for this game. The first pair handles the light equation and the bump mapping effects. The other ones are fairly similar and manage, respectively, the cubemap, reflective and refractive effects.

2.2.1 Multilight

The vertex shader handles the bump mapping (see 3.1) and the positions. The fragment shader handles the lights and displays the textures according to the bump mapping. The lights are managed by generating 2 *uniforms* variables per light. The first is a $vec4$ containing the information about the light position and a boolean indicating whether the light should be rendered or not. The second *uniform* variable is a $vec3$ containing the information about the light's color. The source code of the fragment shader is generated by Javascript functions. The main looks as follows (pseudo code):

```
// variables declarations
precision mediump float;
varying vec3 v_frag_coord;
varying vec2 v_texcoord;
varying mat3 TBN;

uniform sampler2D u_texture;
```

```
uniform sampler2D u_normalMap;
uniform vec3 u_view_dir;

// for each i light source
// uniform vec4 u_light_pos${i};
// uniform vec3 u_light_color${i};‘

void main() {
    // variables initialization
    vec3 color;

    // parsing of the vec4's
    // for each i light source
    // vec3 light_pos${i} = u_light_pos${i}.xyz;
    // float bool${i} = u_light_pos${i}.w;‘

    // for each i light source
    // compute the light as in the exercise session
    // add light contribution to the color vector

  gl_FragColor = vec4(color, 1.0) * texture2D(u_texture, vec2(v_texcoord.x, 1.0-v_texcoord.y));
}
```

2.2.2 Cubemap and Reflective/Refractive effects

We used an HDR image as a cubemap (we created the 6 faces of the cube ourselves). That also allowed us to implement a reflective and a refractive object with this cubemap.

(a) Reflective effect

(b) Refractive effect

It was implemented as seen during the exercises sessions with the following fragment shaders:

```
const sourceRefractionF = ‘
      precision mediump float;
      varying vec3 v_normal;
      varying vec3 v_frag_coord;
      uniform vec3 u_view_dir;
      uniform samplerCube u_cubemap;

      void main() {
        float ratio = 1.00 / 1.52;
        vec3 I = normalize(v_frag_coord - u_view_dir);
        vec3 R = refract(I, normalize(v_normal), ratio);
        gl_FragColor = vec4(textureCube(u_cubemap, R).rgb, 1.0) ‘;
```

```
const sourceReflexionF = ‘
      precision mediump float;
      varying vec3 v_normal;
      varying vec3 v_frag_coord;
      uniform vec3 u_view_dir;
      uniform samplerCube u_cubemap;

      void main() {
        vec3 I = normalize(v_frag_coord - u_view_dir);
        vec3 R = reflect(I, normalize(v_normal));
        gl_FragColor = vec4(textureCube(u_cubemap, R).rgb, 1.0);
      }
    ‘;
```

2.3 Lights

In the first level of the maze, we implemented the Blinn-Phong model (1), combining an ambient light, a diffuse light and a specular light for the sunlight. The light color is a 'pinkish' orange mimicking the twilight.

In the second level of the maze, there is no sunlight, the level is harder since the player has to find the keys in the dark. To be able to still play the game the player emits its own light that is blinking and linearly attenuated with the distance. The light can also move as a sinusoidal function.

In both levels, the keys also emit their own yellow/gold light blinking with a quadratic attenuation.

$$L_{o,\,source}(\overrightarrow{x}) = (A + diff + spec) * att \tag{1}$$

3 Advanced functionalities

3.1 Bump Mapping

The walls of the maze have a sense of depth thanks to the implementation of bump mapping. To achieve this effect, one has to use the tangent space normal mapping. The tangent and bitangent vectors are calculated based on the calculation found in https://learnopengl.com/Advanced-Lighting/Normal-Mapping, these are done during the parsing of the .obj files and passed to the buffer along with the the vertices positions, textures and normals. Then, the TBN matrix is composed in the vertex shader and used in the fragment shader to transform the normals from the normal map in the coordinate system of the world-space.

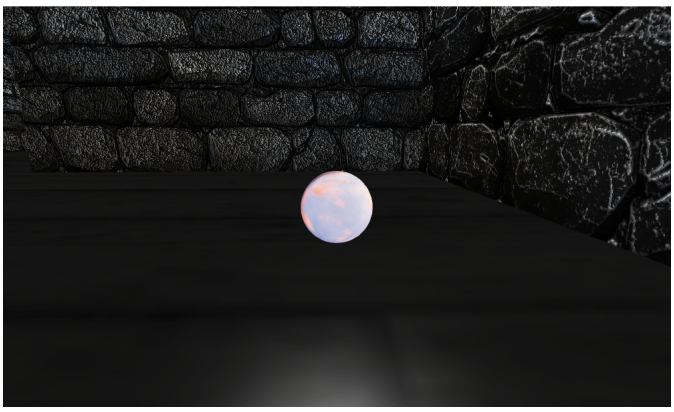

Figure 3: Bump mapping on the walls

3.2 Animations

From the moment the keys are spawned, they rotate upon themselves in a continuous motion.

The doors slide down to let the player in the next level or up to block the passage to the previous part of the maze.

Figure 4: Rotating keys

The player rotate upon themselves according to the movement direction.

3.3 Collision detection

The player cannot get through the walls of the maze, in any direction. At each frame, the game computes the next position of the player and checks if there is a collision at this position. It does so by checking for each game object if the positions are the same and the object is an *obstacle* object. The matching of the positions is defined differently for each object. For example, if the object is a cube, the matching function will check if the position is contained in the box. When a collision is detected, the position is not updated to the next position.

3.4 Physics

The player can use the spacebar to jump. The jumping movement can be combined with any other movement. In the rolling movement, the player is considered to be in a Uniformly Rectilinear Motion (URM).

In the jumping movement, the player is considered to be in a Uniformly Accelerated Rectilinear Motion (UARM). Thus, its position in the Y-axis follows the following equation.

$$y = y_0 + v_0 \cdot (t - t_0) + \frac{a \cdot (t - t_0)^2}{2} \tag{2}$$

That translates into the code like below:

```
nextPos = [position[0], startJumpPos + 5.0 * delta + (-9.81 *
   (delta)**2) / 2.0, position[2]];
```

The acceleration is set as $-9,81m/s^2$ to mimic a real world descent. The initial velocity v_0 is set as $5m/s$.