MazeGL

A procedurally generated maze made with WebGL and Three.js

Samuele Vittorio Fusaro

July 2018

The maze

The project is based on a procedurally generated and potentially infinite maze. The base idea is a camera traveling across a system of hallways and crossroads which is randomly generated on movement.

A lighting system which generates along the maze has also been developed.

Three.js

The Three.js library was used, which helped a lot the programming phase and permitted to focus more on the development of the custom shader.

It was used mainly for:

- mesh creation and management;
- uniform variables creation and management;

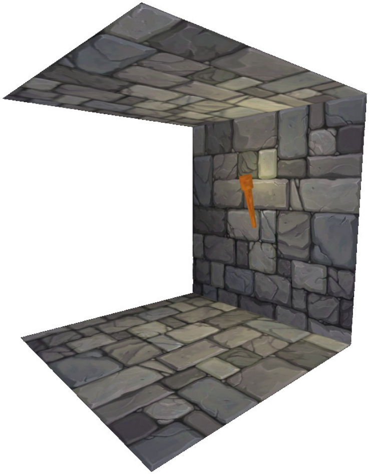
The Javascript code

The 3D space

The whole X-Z plane area is split in a matrix of **blocks**, which are intended as a cube composed by a floor, a ceiling and 4 sides with possible walls. The 3D space is then "quantized" (the integer part of the division between camera coordinates and block dimension) on a grid where the single element is a block,

Since the camera is "non flying" just the **X and Z coordinates** are needed. The Y coordinates are used just to set the height of camera and meshes. So, each block can be found with its X and Z coordinates based on the block dimension.

Depending on which of the 4 walls are placed on each block, the maze will develop on different ways.



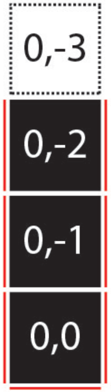
A block, with floor, ceiling and just one wall, in this example.

The Javascript code

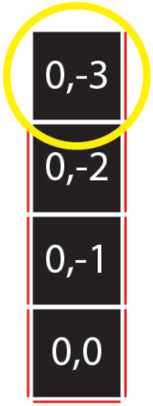
The algorithm

1. The starting step is the generation of a series of blocks with just walls on the sides, creating an hallway. At the end of this hallway there will be an empty space.
2. Then the first crossroad will be generated on that empty space;
3. On each open side of the crossroad an hallway of random length will be generated;
4. The end of each hallway is the spot where a new crossroad will be generated;
5. And so on...

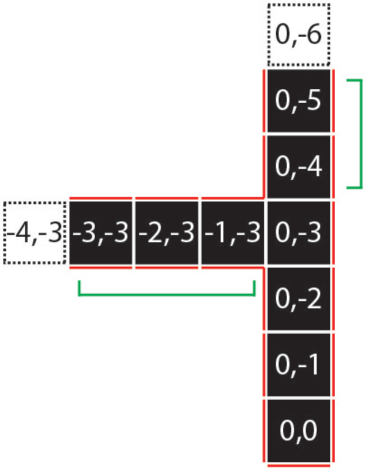
- In this implementation a crossroad is intended as a block where, except for the entering side, the presence of the other 3 walls is random (it could be also a dead end);
- The coordinates of each empty space at the end of a hallways are saved in a list to keep track of next generation spots. A second list is constantly updated to hold the nearest empty spots.
- The crossroads have just 90 degrees turns for simplicity.



Step 1: the red lines are walls for each block, the dotted square is an empty spot where to create a new block.

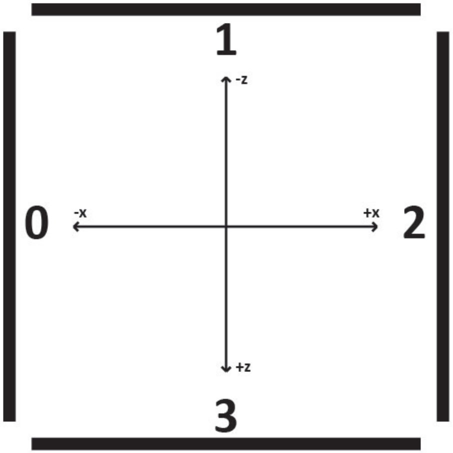


Step 2: the new crossroad block is created, with random generated walls.



Step 3: for each free side of the new block, the hallways are created. New empty spot are available.

Each wall is identified by a number as shown in the image. To describe the structure of each block a 4 element array is sufficient: if the n-th element is 1, there is a wall on the n-th side.



The Javascript code

The rendering grid

In order the improve performance, just the meshes in a certain distance from the camera are rendered.

To define which blocks are within the rendering area, the **point distance formula** was used. If the distance between the coordinates of the block where the camera is and a generic block is fewer or equal to d then the block is rendered.

$$d = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$$

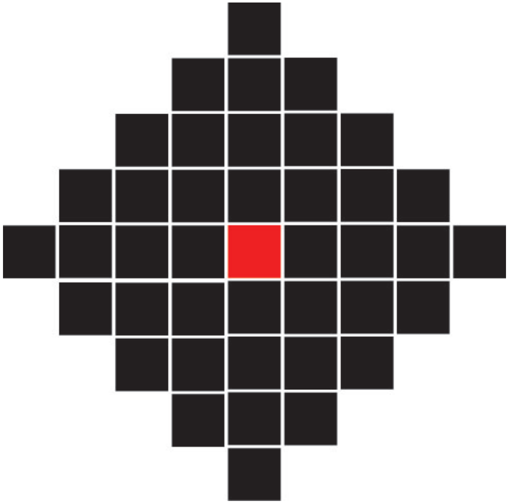
Where (x_0, y_0) and (x_1, y_1) are the coordinates of the two blocks.

When entering a new block the grid must be computed again, and the block which will result outside the grid deleted.

As mentioned, when a block is not in the rendering area it will be deleted. If, after the camera movement, the block is renderable again, it will be reconstructed.

To keep track of the generated blocks a *mazeMap* is used. It is a simple **Javascript object** which takes the X and Z coordinates of a block as key and holds the walls array as value. This way when reconstructing the grid, first it will be checked if the block is already in *mazeMap*. If yes, it will be reconstructed. If no, it could be generated randomly.

Since the blocks coordinates are integer values, and the point distance formula is used, the rendering area will result like this:



The Javascript code
labyrinth()

labyrinth() function extracts the nearest spot where to generate a new crossroad block and, after all the needed controls, generates the crossroads and all the hallways next to it.

- In the first part the function checks if there is any close spot where to create a block.
- It will also checked if the nearest empty space is inside the rendering grid.
- Then the crossroad block will be generated, randomly choosing the free walls.
- Once the block is rendered, for each free wall a hallway of random length between 1 and *gridDim* is generated.

The Javascript code

genBlock()

This function, given the X and Z grid coordinates and the binary array for the walls, will generate a block.

First, accordingly to the walls array, the walls, floor and ceiling meshes are created.

Then the meshes are placed and rotated in the correct position in the 3D space using their grid coordinates and the blockDim value.

For example for the left wall:

```
var object = new THREE.Mesh(plane,wallMaterial);
object.position.set(
       -blockDim/2 + (pos[0]*blockDim),
       blockDim/2,
       (pos[1]*blockDim)
);
object.rotation.set(0,Math.PI/2,0);
```

The Javascript code

animate()

This function is called to constantly draw the scene, using the *requestAnimationFrame(animate)* Javascript command.
Here the camera position is constantly checked to detect if a new block is entered. When that happens the following actions are made:

- the blocks inside the rendering grid are rendered;
- the list of the empty spots where to create new blocks is reordered based on the new camera position;
- the *labyrinth()* function is called, in order to generate the new blocks in the available spots;
- the blocks which are now outside the grid are deleted.

The shaders

Instead of the built-in materials that come with Three.js, a custom shader was developed using the GLSL language, to model the materials used for the walls and then the lights effect.

Pointlights were used. They are managed as Three.js objects, which takes care of placement and variables handling.

Vertex shader

The vertex shader is pretty standard. It computes:

- the *gl_Position* vector using the *projectionMatrix* and *modelViewPosition*.
- The *vecPos* vector, used in the Fragment shader to compute the dot product for light calculations.

Fragment shader

Here, all the **light** and **fog** computations are made.

Using the **uniforms** containing lights parameters provided by Three.js and passed to the shader, a *struct* is defined to hold this variables.

```
struct PointLight {
       vec3 color;
       vec3 position;
       float distance;
};
uniform PointLight pointLights[NUM_POINT_LIGHTS];
```

Three.js also provides the **normals** for every point, needed for computations, to the shader.

Fragment shader

Light dot product

For each pixel the light which hit the point is computed this way:
A vector *addedLights* is initialized to 0.
For each light, the dot product between the normal and the vector from the light point and the texel point is computed, as follows.

```
max(0.0,dot(-lightDirection,vecNormal))
```

max is used to cut off values below 0.

Fragment shader

Light attenuation

As resulted in the development process, the dot product itself is not sufficient to create a correct light model in a 3D space.
The fact that light dims as it travels in space is not considered until now.
A light attenuation formula is the needed to represent the correct light value for each point light in the area.

$$\frac{1 - distance^2}{radius^2}$$

Where *distance* is the distance from the considered point to the light source, and *radius* is an arbitrary value.

Other light attenuation formulas have been tried in the development phase such as:

$$\frac{1}{1 + 0.1 distance + 0.01 distance^2}$$

but it was found that the best results were given by the aforementioned formula.

The final light computation is the following:

```
for(int l = 0; l < NUM_POINT_LIGHTS; l++) {
        lightVec = vecPos - pointLights[l].position;
        attenuation = clamp(1.0 - length(lightVec) *
            ↪ length(lightVec)/(blockDim * blockDim *
            ↪ 1.8),0.0,1.0);
        vec3 lightDirection = normalize(lightVec);
        addedLights.rgb += clamp(max(0.0,dot(-
            ↪ lightDirection,vecNormal)) * pointLights
            ↪ [l].color * attenuation,0.0,1.0);
}
```

Other mentions

Other relevant aspects of the application are:

- Wall collision;
- Torch model creation;
- Fog;
- First person camera;
- Parameters;

Experiments and conclusions

The application was tried in 2 different ambients.
The performance are quite good and permitted a good experience for the user.

- With a medium-end laptop graphic card (nVidia GeForce 940MX) the application runs between 28-35 FPS;
- With a desktop graphic card (AMD HD 7890) the application is pretty often stable at 60 FPS;

The application has been tried also for quite long time, in order to test the mesh creation and deletion mechanism and the map structure saving feature in the long time.
Even when the map reached a quite big size the application ran smoothly.