SAPIENZA
UNIVERSITÀ DI ROMA

INTERACTIVE GRAPHICS - FINAL PROJECT

MineGL

Armando Nania

Jose Luis Bustamante Julca

September 2020

Introduction

MineGL is an exploration 3D browser game realized as final project for Interactive Graphics course. In it, the player can explore the world, encountering some animated creatures during the day and night. It is based on a popular game called Minecraft, where the goal of the user is to explore the world. The project has been tested on Google Chrome and in order to properly run it, the user needs to have a web browser that supports WebGL 2.0 and an up to date javascript interpreter since classes are used in the source code.

Implementation

The project is realized in WebGL 2.0 (Web Graphics Library). It is a Javascript API for rendering interactive 2D and 3D graphics within any compatible web browser without using plugins.

Third-party libraries:

- *simplexNoise.js* : used for random generation of the map, library which allow us to have a lower computation complexity during a generation of a well-defined and continuous gradient (almost) everywhere.

As introduced before, this is a program inspired by the famous video game called Minecraft where the player controls a character in a randomly generated world starting from a seed. This is a world made of cubes, called Voxel and later we'll see how we exploited this kind of structure for the development of our physics engine. MineGL does exactly the same thing: when the web page starts, a javascript function, *Date.now()*, is called in *main.js*. The returned value is then used to set the seed of a multidimensional noise function, the Simplex Noise function:

const openSimplex = openSimplexNoise(Date.now());

This particular function has the characteristic of being a smooth function. This means that we can use it during the terrain generation, where such behavior is needed. Using a normal RNG, in fact, there would be a sudden change in terrain values, making the scene more like a hellscape rather than a natural scene. The classic RNG function (Math.random()) is used instead where a smooth change of values is not necessary, for example in the code related to artificial intelligence, where which animation to execute next is decided.

As for the generation of the terrain, the original idea was to create an infinite world. For this we created a chunkT function that generates a piece of terrain, and we used this function to create 9 objects of this type, arranging the 9 terrains in a three-by-three conformation. What we intended to do was to dynamically generate new terrains as the player explored the world, while simultaneously erasing old unused terrains. While we haven't been able to implement this in time, the code still has the basic structure to be able to add this functionality. As we will see, the abstraction necessary for the development of the undeveloped feature has complicated not a little the writing of the rest of the code. (but it will be implemented in the future…)

Structure

The code has been splitted into several files, each one with a specific task.

- index.html: File which contains both vertex and fragment shader. To open the game, user has to run the HTML file.
- main.js: the file that manages the scene and the generation of the world.
- horse.js: the file that contains all the functions used to create and animate a generic horse.
- spider.js: the file that contains all the functions used to create and animate a generic spider.

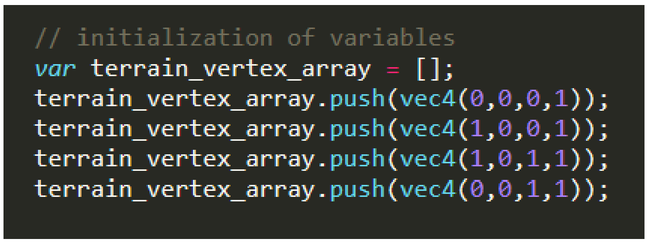
Scene

The rendered scene is divided into two parts: the terrains and the entities. These two types of objects are rendered differently, but they use the same program. Since terrains use instances, while entities do not, we decided to arrange everything using two model matrices: one matrix of type "attribute", used during instancing, and the other of type "uniform", used in rendering of the entities. When rendering of an object, the mismatched matrix is set to the identity matrix. Although it is not the most elegant solution, it is certainly the fastest to implement when you have two different codes to integrate. And best of all, it works. A more efficient solution would have been to change the rendering of the entities, to make them also use instancing. This would not only make the code more consistent, removing the need for two model matrixes, but also increase the overall program performance. This is one of the first points to review for a possible future development of the project.

Terrain

The terrain, in Minecraft, is a set of cubes of earth, rock, snow and water that you can walk/swim on/in. In our version of Minecraft in WebGL, since these cubes are fixed and cannot be removed, we were able to simplify the scene to render. In particular, we considered each terrain as if it was a grid of cubes. For each 'x' and 'z' value of the grid we generated a 'y' value using the simplex noise function. Everything is stored in an array called 'map'. From a coding point of view, terrains are javascript objects of the class-function "chunkT".

Terrains are instances of the chunkT function. Each terrain has several properties, including the 'map' matrix. In addition to the terrain map, there is a cloud map, called 'cloud_map'. This map, unlike the terrain map, is a Boolean matrix. Each element of the matrix indicates whether or not there is a piece of cloud in that box in the sky. The y-value of the clouds is constant for all clouds, and is equal to 80. After generating the two matrices 'map' and 'cloud_map', it is necessary to decide what to actually render. The idea used here is to render only squares, meaning to render only gl.TRIANGLE_FAN with only four points:



```
// initialization of variables
var terrain_vertex_array = [];
terrain_vertex_array.push(vec4(0,0,0,1));
terrain_vertex_array.push(vec4(1,0,0,1));
terrain_vertex_array.push(vec4(1,0,1,1));
terrain_vertex_array.push(vec4(0,0,1,1));
```

This simple square is translated and rotated in the scene, in order to make a something similare to a "dress" that fits perfectly with the surface of the cubes of the ground. Then, for each cube, one of these squares will be rendered, plus another square for each visible side part of that cube, and any possible visible cube below it. In this way, we are sure to render the minimum necessary number of polygons, so that there are no "holes" in the ground. To render a generic square using the aforementioned points, a model matrix is associated for each square to be rendered. The model matrix is responsible for rotating and translating all the points of the square and placing them in the desired position. So we actually have a long list of matrices, called instance_matrix_array, and for each of these matrices we will always use the same 4 points. This style of rendering is called instancing, and it turned out to be the best performance improvement we had while trying to optimize the code. In fact, we initially used the *gl.drawArrays()* function, and we had fps drops while rendering 15k polygons. Using the *gl.drawArraysInstanced()* function, instead, we are able to draw more than a million polygons without any problem! However, this forced us to use instancing for other types of buffers as well, including the texture_coord buffer (our textures use the technique of texture combination called texture atlas). The alternative would have been to divide the matrix list buffer into several lists, one for each type of texture to be used. While this alternative could have provided a performance boost, we decided not to use it, as the code is already very complicated in itself.

To end terrain discussion, we must add the fact that the textures of the vertical squares are different from the textures of the horizontal squares. Also, to simulate snow, we set different textures for the very top squares, those with 'y' greater than snow_level. During the generation of the 'map' matrix, then, we set a minimum limit of the generated values, equal to water_level. In this way we were able to generate lakes and rivers as well. Each terrain also has properties that indicate the offset x and offset z. These two properties are used to identify the unique position of a terrain relative to the world frame. This is necessary as the world is divided into 9 chunks. So each chunk must know where it is and what its neighboring chunks are, in order to generate vertical walls even between two different chunks. In addition, the terrains are designed to be managed dynamically, even if we actually don't do that.

Models

For the creation and animation of the animals inside the world, no external libraries have been used. A classical hierarchical model was computed as represented in the figure:

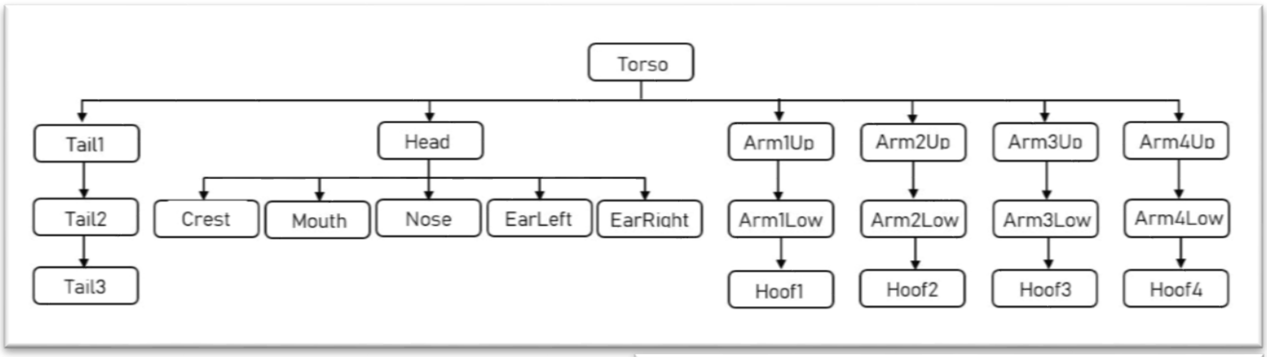


Figure1. Hierarchical model of a horse

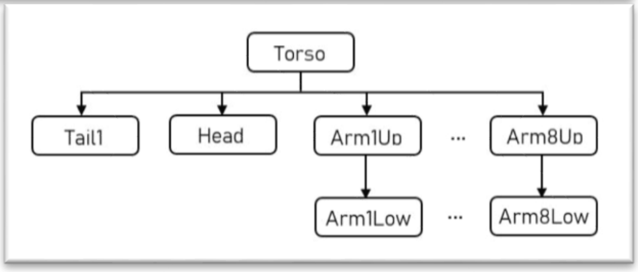


Figure2. Hierarchical model of a spider

Each animal is treated as an 'object', with its own properties. In the class horse.js, each one has its local variables since these are the characteristics that differentiates each horse from another (e.g. position in the world, values of angles of its body). The global variables represents the common characteristics, such as dimensions of bodies or numbers of nodes for every horse.

Every animal has its own animation, which is chosen randomly from 3 available animations (for the horse). They are:

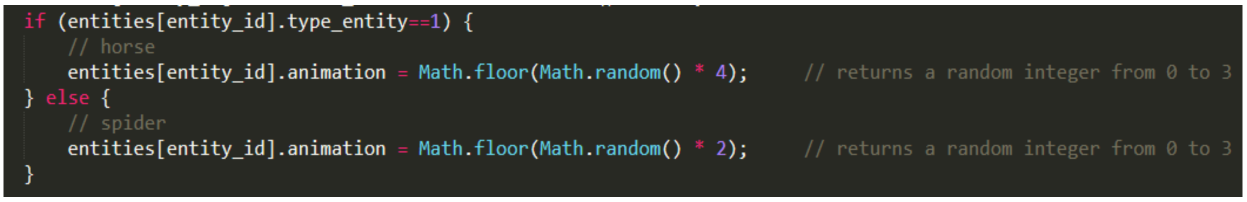
-walking_animation() : Simple animation where the horse moves in a direction given by the file main.js (computed randomly).

-rearing_animation(): Animation where the horse stands up on its rear legs and kicks in front with its front legs.

-eat_animation(): In this animation the horse

There is another animation called standH() which resets all the local variables for the selected animal, so it can be ready to start another random animation. All these animations have been created for this project and no one have been imported from outside.

In the file main.js an array of animals is created, the animals are ‘entities’ and the type of animal selected is given by entity_id. Since they are ‘objects’, for every animal of the array a random animation is called as follows:

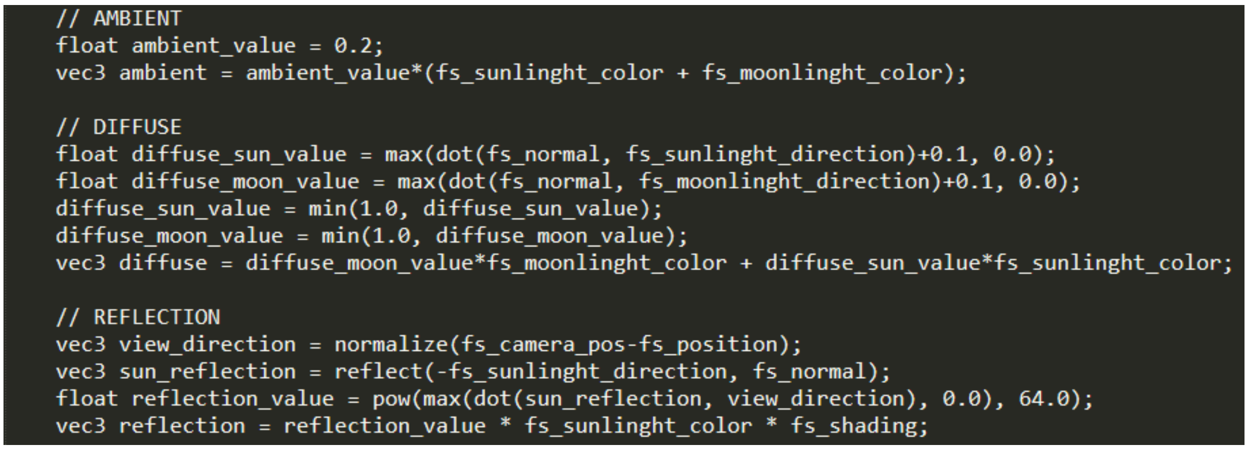


```
if (entities[entity_id].type_entity==1) {
    // horse
    entities[entity_id].animation = Math.floor(Math.random() * 4);     // returns a random integer from 0 to 3
} else {
    // spider
    entities[entity_id].animation = Math.floor(Math.random() * 2);     // returns a random integer from 0 to 3
}
```

Since we have 3 animations, a random integer from 0 to 3 is created, and if the number 0 is obtained, no animation starts and the animals stands in its position.

Lights

In the fragment shader of html file, a Phong shading interpolation technique was used, combining the Phong interpolation and the Phong Reflection model. To follow the Phong’s equation and Ambient, Diffuse and Specular values are computed as follows:



```
// AMBIENT
float ambient_value = 0.2;
vec3 ambient = ambient_value*(fs_sunlinght_color + fs_moonlinght_color);

// DIFFUSE
float diffuse_sun_value = max(dot(fs_normal, fs_sunlinght_direction)+0.1, 0.0);
float diffuse_moon_value = max(dot(fs_normal, fs_moonlinght_direction)+0.1, 0.0);
diffuse_sun_value = min(1.0, diffuse_sun_value);
diffuse_moon_value = min(1.0, diffuse_moon_value);
vec3 diffuse = diffuse_moon_value*fs_moonlinght_color + diffuse_sun_value*fs_sunlinght_color;

// REFLECTION
vec3 view_direction = normalize(fs_camera_pos-fs_position);
vec3 sun_reflection = reflect(-fs_sunlinght_direction, fs_normal);
float reflection_value = pow(max(dot(sun_reflection, view_direction), 0.0), 64.0);
vec3 reflection = reflection_value * fs_sunlinght_color * fs_shading;
```