Interactive Graphics - Homework 1

Marco Lo Pinto, 1746911

May 1, 2021

The homework uses Object Oriented Programming Syntax in JavaScript (ES6), so to have modularity and to reuse of code through inheritance. It also exploits promises with async/await syntax to load the images. This project has been tested both on Google Chrome (version 90) and Edge (version 90). It was also tested on Firefox (version 86). It is important to create a local server in order to bypass browser restrictions.

The Homework1 folder structure is organized as follows:

- **tools/**: In this folder there are the classes to implement the camera, the lights, the shapes and the objects.
- **textures/**: All the images to load are stored here.
- **shapes/**: In this folder there are all the shapes generated by extending Shape.js in the folder tools.
- **Homework1.js**: Contains the main class of this project.
- **Homework1.html**: The page of this project which contains also the fragment and the vertex shaders.

To generate the principal object, firstly it is extended the Shape.js class to create the shape: using the method triangle() it receives three vertices (namely a, b and c) and computes the normal as the normalized cross product of two vectors generated by the difference of the vertices (this is because the vectors are complanar with the triangle). Exploiting the direction in the 3D space of the normal, it is used to compute the texture coordinates: in essence the code checks at which face the normal is pointing at inside an imaginary cube surface (using spherical coordinates). After that, it is called the function convertToTextureCoords() to generate the texture coordinates giving the vertex chosen and two of the three indices calculated using the logic described above (for example, if the face is the "top", it is used the x as first coordinate and y as second). If the object is scaled above the interval [-1,1] (or [0,1] in texture coordinates) the image will repeat generating a pattern (see the terrain as an example).

In the Homework1.js class, there are two main methods to compute the barycenter: the first method calculates the barycenter taking into account the area of each triangle and its position in the 3D space and computing the average point (deprecated), while the second one uses the definition of geometric barycenter to compute it. There is a button in the page that toggles the rotation around the defined center of the object and the barycenter.

The view and the projection matrices are managed in the Camera.js class: exlpoiting the perspective() and the lookAt() in MVNew.js, firstly the camera class is initialized at position

(0,0,0) and rotation (0,0) when created. Then there are functions to rotate and move the camera by a desired interval delta (dx,dy,dz for the movement and dthetha and dphi for the rotation). The rotation is computed using spherical coordinates to simulate the visual movement of the eye and/or head of an human; the same approach is used to define the movement. To make it easier to move around the space, the controls are identical to 3D video games: w,a,s,d to respectively walk forward, left, back and right with respect to the point where the camera is looking at in the x-z plane, space and shift to respectively fly up or down (in the y-axis). These controls are not only via keyboard, but there are also buttons on the screen.

To compute the cylindrical light, firstly it is used the class Lights.js to create three new lights inside the cylinder: each light must have position, ambient, diffuse and specular properties to be defined. After that, these lights are computed on the vertex and in the fragment shader using the logic of the Phong model, passing four arrays, each of them is one of the four properties defined above and with dimension equal to the number of lights into the scene. To compute them correctly, it is important to remember all the spaces in which we are working on:

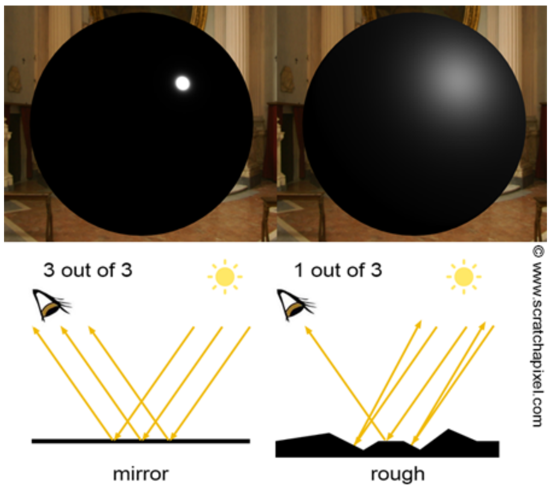
Tangent Space <-> Model Space <-> World Space <-> View Space <-> Clip Space

If it is needed to move from a space to another, it is necessary to use a base change matrix, such as model, view, projection and normal matrices. For example, if the position of the vertex is on the model space, to move from here to the world space: ModelMatrix*pos.

In the shaders, every component is transported to the view space to make all the necessary computations (in the code every step is explained with comments).

The cylinder shape is created by extending Shape.js as for every other objects; it receives the radius of the cylinder, the height and the number of vertices of the top and bottom face (less vertices equals more approximation). The cylinder object instance of the class ObjectMaterial.js has a parameter, called emissivity, that simulates in the shader the effect of a light source object (like a lamp or a moon; reference used: Edward Angel, computer graphics 8th edition, chapter 6.7); the lights can be turned off and if that happens, also the cylinder is turned off.

For what concern the material properties of the principal object, it is assigned a bright blue color for the ambient to have correspondence with the texture. Regarding the shininess, instead, because values between 100 and 200 correspond to surfaces like metals and between 5 to 10 is for plastic surfaces and because the material that it is simulated is ice, the chosen value is 80 (the terrain, instead, is 10). The effect of light on different surfaces is summarized below:



To show the differences between the per-vertex (Gouraud shading) and per-fragment (Phong shading) shading models with the light, there are three buttons on the page: one for the vertex computation, one for the fragment computation and one for the fragment computation with the bump map on the object; it is pointless to compute the bump map in the per-vertex strategy: this is because it finds the average normal at each vertex.

The bump map can be generated in various ways on the Shape.js file: two main interesting techniques of doing that are by creating a random bump (noise), which can make a more or less rough surface, depending on the min and max parameters passed on the function, and the bump generated by the image. The latter is done by using the function createImageGrayScale(), which uses a canvas element to capture all the image data in an unsigned int8 clamped array (dimension: height*width*4 bytes) and then processes every rgb component as the average of the three (excluding alpha), generating as a greyscale matrix array. After that, the result is passed to computeBump() that will generate the bump map and load it in an array.

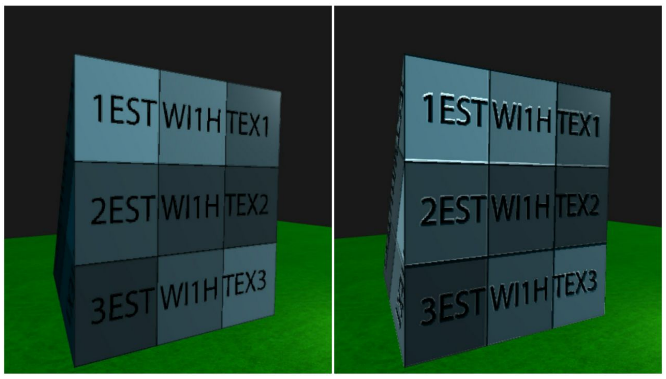


Figure 1: Texture without bump map (left) and with bump map generated with createImageGreyScale (right)

The general program execution flow is as follows: when instantiating the class objectHomework1, it accepts as parameters canvasID, which is the id of the canvas object, and the imageTextures dictionary. After initializing some flags for the barycenter and for the terrain and after instantiating the camera class, it gets the drawing context of the canvas and loads all the two main programs: one for per-vertex and one for per-fragment computation. After that, using the class Lights, it generates the three lights that will be inside the cylinder and also an extra light that simulates a very far away light (useful to see the light when the cylinder is off). The next step is to create the shapes for the ObjectMaterial class: among all the methods that are present in the Shape class, the triangle method is one of the most important to analyze. Other than doing what already said at the beginning, it also computes the tangent for the bump map.

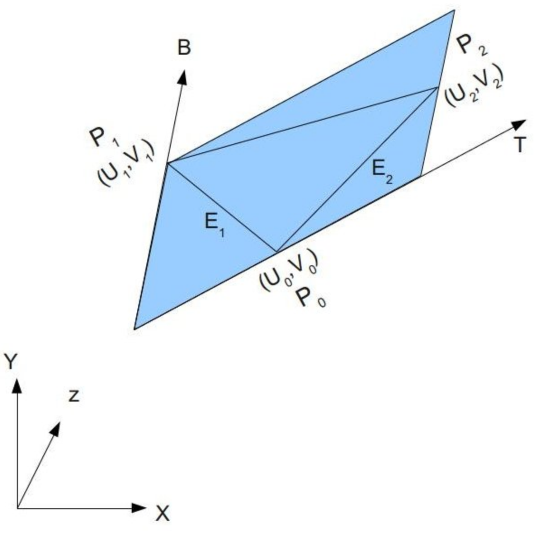
Starting from the texture coordinates generated (namely P0, P1 and P2, see the image on the next page), if the shape has a bump map, then knowing that the two triangle edges E1 and E2 can be written as a linear combination of the tangent and of the bitangent (see figure above for reference):

$$E_1 = (U_1 - U_0)T + (V_1 - V_0)B$$

$$E_2 = (U_2 - U_0)T + (V_2 - V_0)B$$

This can be written as a product of matrices:

$$\begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$



by using the inverse of the second matrix, it is calculated and normalized the tangent (the binormal will be computed on the shaders).

After creating the shapes, the objects that will be drawn into the scene are: the main object, the cylinder with emissivity and the floor to have a better visualization of the light.

When the render method of the class objectHomework1 is called, after clearing the buffers with gl.clear(), the camera class generates the view and the projection matrices that will be used in the shaders. Next, the Lights class instance will return all active lights (this is because the lights can be turned off) and passes them to each object via the method computeLights(): this creates all the necessary informations for each light source, such as the ambient, diffuse and specular product between the light and the material, the light position and so on. Finally, each object calls the method renderObject() which receives as parameters the model and the view matrix: here all the necessary buffers are binded and activated. One important note is that the model matrix passed will be the first element on the resulting model matrix generated by render object: this is because here it computes the rotation and the position of the object, which are parameters that can be set using the methods setPosition(), setRotation() and setCenterOfRotation() (the latter useful for the barycenter computation).